DTIC FILE COPY

AFOSR-TR- 87-0690

OPTICAL SYMBOLIC PROCESSOR FOR EXPERT SYSTEM EXECUTION

QUARTERY TECHNICAL REPORT

December 1, 1986 to February 28, 1987

Sponsored by
Air Force Office of Scientific Research
and
Advanced Research Projects Agency (DOD)
ARPA Order No. 5794
Contract #F49620-86-C-0082

AD-A181 833

DTIC
ELECTE
JUN 24 1987
S D

Prepared by

Matthew Derstine
Aloke Guha
Subra Natarajan

Honeywell Physical Sciences Center
10701 Lyndale Ave. S
Bloomington, MN 55420

Submitted: Matthew Derstine
Matthew Derstine, Principal Investigator

Approved: Anis Husain
Anis Husain, Section Head

Approved: D E Fulkerson
David Fulkerson, Department Manager

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DTIC
This technical report has been reviewed and is approved for public release IAW AFR 190-12.
Distribution is unlimited.
MATTHEW J. KERPER
Chief, Technical Information Division

Approved for public release;
distribution unlimited.

87 5 21 051

SECURITY CLASSIFICATION OF THIS PAGE

AD-A181 833

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 87-0690
6a. NAME OF PERFORMING ORGANIZATION: Honeywell Inc. Physical Sciences Center; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: AFOSR / NE
6c. ADDRESS (City, State and ZIP Code): 10701 Lyndale Avenue South, Bloomington, MN 55420	7b. ADDRESS (City, State and ZIP Code): Bldg 410, Bolling AFB, DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: same as 7a; 8b. OFFICE SYMBOL (If applicable): NE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: #F49620-86-C-0082
8c. ADDRESS (City, State and ZIP Code): same as 7b	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2305	(DARPA) B1	

11. TITLE (Include Security Classification): Optical Symbolic Processor for Expert System Execution

12. PERSONAL AUTHOR(S): Matthew Derstine

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Quarterly Technical	FROM 12/1/86 TO 2/28/87	87/04/23	51

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Optical Computing, Symbolic Computing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes a novel architecture for an optical symbolic computer. The symbolic architecture, Symbolic Processing Architecture in Optics (SPARO), is designed for executing functional language programs using combinator graph reduction. SPARO is designed with the goal of exploiting the available fine-grained parallelism of both combinator graph reduction and primitive optical operations. A planar array of processors, communicating by messages over a network, provides the processing power of SPARO. The finite state machine of individual processors is expected to be implemented using symbolic substitution techniques, while gateable interconnects would be used for realizing data movements between the processor and the network. We propose a simple register-based network that would enable multiple messages to be delivered concurrently. It is shown that the architecture can be easily scaled to accommodate large combinator graphs. The detailed control sequence required for processing within the nodes and for messaging are shown as macro-instructions that would be executed by each processor in SPARO. These macro-instructions can be further translated into simpler symbolic substitution rules

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UUUUU

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. C. Lee Giles	202/767-4933	NE

87 5 21 051

1. INTRODUCTION

In recent years there has been a great deal of interest in combining the extraordinary communications power of optics with symbolic computing techniques to develop a highly parallel optical computer. Advances in parallel computer architecture and development of digital optical components and systems have laid the groundwork for such a computer. In this report we describe the development of an optical architecture designed for symbolic computing.

In previous work [1] we examined logic and functional languages to determine which had characteristics more amenable to optical implementation. We found (cf. 2nd Tech. Report) that functional languages were a better fit than others and thatcombinator graph reduction (CGR) was the appropriate computational model. We also examined symbolic substitution and other optical computing techniques to determine which could provide the most power and flexibility. We decided that symbolic substitution coupled with some form of data movement scheme was the best scheme to employ.

In this quarter we have continued the development of an architecture based on symbolic substitution and combinator graph reduction. This architecture, SPARO (Symbolic Processing ARchitecture in Optics), was developed to show how optical devices and systems might be utilized to construct a scalable parallel optical computer. SPARO is not just a processing element, but rather a total computer system. It supports primitives for the functions required for symbolic computing including memory management and recursion.

We begin the description of SPARO by first giving an overview of its operation and its components in Section 2. The principles of its operation and its morphology are described in Sections 3 and 4. Section 5 describes how CGR is performed. The optical implementation of thecomputer is described in Section 6. Section 7 describes issues which we have not yet addressed, and in Section 8 we draw conclusions.

2. OPTICAL ARCHITECTURE FOR COMBINATOR GRAPH REDUCTION - OVERVIEW

In this section we provide a broad overview of the architecture we have proposed, saving the details for the later sections. We give an outline of the processor and the network that constitutes SPARO.

SPARO uses a physically non-distributed architecture where the compiled program, i.e., the source program translated into the combinator graph, is considered to be stored in a single optical plane or an array of optical pixels. All operations such as graph traversal, data moves during combinator reduction, and the evaluation of functions can be considered to be executed in this same plane.

To understand the operation of SPARO, we first describe how it is partitioned into functional elements and how those elements perform combinator reduction. In later sections we more fully describe how those functions could be implemented using optical devices and systems. The following subsection describes the basic functional elements of the SPARO processor.

2.1 Processor Design

DTIC COPY INSPECTED

Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

One of the major obstacles overcome by the SPARO architecture is the difficulty of implementing addressable memory with optical components. This is a major accomplishment since the design of the processor for CGR is intimately tied to the design of local memory in a non-distributed architecture. The memory design, in turn, reflects the nature of data structures used for CGR. We will therefore describe the data structures and the associated memory structure in SPARO and show how they dictate the processor design.

The computations in CGR and, in general, symbolic processing, are very memory intensive. To guarantee efficient computing in SPARO, the choice of the memory interconnection topology becomes important. Although many different memory organization topologies have been proposed and implemented, the choice was difficult because most topologies are not efficiently implemented in optics. The criteria for choosing the memory structure was based on the following observations.

First, there are no efficient ways to implement addressable memory optically. If an optical memory were to be constructed based on the structure of electronic memories, it would be grossly inefficient because of the time required to decode the addresses, ([4], 2nd Tech. Report) since many logic stages are required for a 1-of-n decoder. Consequently, optical memory must take the form of registers and of small easily decoded addressable memory.

Secondly, the density of general interconnections that can be made are limited [15]. This limitation constrains the optical connections to be highly regular and precludes the use of dynamically reconfigurable optical interconnects to represent the connections between pieces of data.

Finally, the interconnection scheme between related data in the memory must be very general to be efficient since the links between nodes of the graph will change often. It is not practical to move data items and have proximity indicate connections since the movement of data is inefficient compared to pointer manipulation. Proximity is also excluded as a means of representing the data structures since the vertices of an abitrary graph cannot be represented in a two-dimensional structure.

Using these criteria, we examined different memory topologies. In each case the direct data moves and modifications of connectivity for the graph compromise performance. It is preferrable to have all data in fixed locations while pointers are emulated for data access. Such a scheme reduces the volume of data movement, which is otherwise expensive.

This suggests that the architecture employ fine-grained processing elements or nodes (as called henceforth because of their analogy to nodes of the graph and to nodes of a network) in a non-distributed architecture [1]. The processing elements are very tightly coupled, and relatively simple communication, a few data words, is used. Also the memory is not separated from the processing elements and thus can be considered to be a part of the nodes. This structure meets the requirements imposed by optical implementation by reducing the need for addressable memory while still exposing the parallelism in CGR.

If our architecture is to scale to a large number of processing nodes, then the pointers, or addresses, will typically require more than 20 bits. It will be shown

later that this means that a typical node will require about 1000 bits of storage. If optical elements are employed which have 1000 x 1000 pixel elements, then an efficient representation is a linear array of nodes. This is important since both the connections between nodes as well as the contents of the nodes can be represented. Using a linear array as described is also attractive because the optical interconnections will be more regular.

Each of the processing nodes uses only registers as memory because of the difficulty in implementing typical forms of addressable memory with optical elements. To perform graph reduction, the nodes operate by messages. Thus processing in a node is activated on receipt of a message while it sends out messages over a network to initiate other graph reduction operations.

A separate interconnection network is used to pass messages (with data) from node to node. The interconnection network could be implemented in any fashion insofar as it allows the transfer of multiple parallel messages. We now discuss the structure of the processor and memory.

2.1.1 Structure of processor and memory

The combinator graph (CG) has a very special structure. Each node in the graph has an 'arity' or out-degree of two. The only difference between the CG and binary tree is that nodes in the CG may be shared by more than one parent node (all nodes can only have two child nodes). This limited arity of nodes in the CG suggests a direct mapping of the graph onto the processing elements. The advantage of this mapping is that each node (as memory) only needs a limited number of memories since it only has to maintain information about itself, its two children and its parents (the case of multiple parents is explained in Section 3). As a processing element it also contains information regarding its state and some very limited working storage. The nature of architecture requires that the external interconnection network transfer data to and from nodes in the array. Locations of interconnected nodes cannot be guaranteed to exhibit locality in the array, due to the nature of graph mutations in CGR. We have therefore partitioned the processor and the communication into two separate functional parts to solve the connectivity and data movement problem. The next subsection explains how the processor array and the network can emulate a dynamic memory.

2.2 Network Design

The functionality of the network is determined by the nature of operations required for CGR. The gross functions that are required as capabilities of the network are graph traversal and data movement. These capabilities in turn require that the network must be able to locate nodes in the array by their node numbers, and be able to read/write data in and out of the data fields (e.g., node value or node number). Another required feature of the network is the parallel access of nodes in the plane. Since arguments of a strict function (i.e., a function that needs the values of all its arguments to compute) can be evaluated in parallel [11], the network should be able to accomodate multiple data movements in disjoint portions of the CG. Efficient parallel access of the nodes in the graph also implies that there should be low or at best no contention in the use of the network. The problem of network contention, which arises in this network, will be addressed by the network processor described in the Section 6. In the intervening sections, the

network can be considered to be a resource that quickly transfers several words of data from one node to another.

2.3 Optical Implementation of SPARO

Examination of optical computing structures have revealed that the most promising concept for realizing computing is symbolic substitution [3]. We have found, however, that it cannot be easily used for simple data movement. A simple solution would be to use symbolic substitution for the control and logic operations, while using other techniques to move data between registers and nodes.

The technique for moving registers and nodes that we are currently examining is to provide limited, regular interconnects that have optical logic gates at the inputs and that are controled by pixels of the symbolic substitution system. Thus by enabling or disabling the logic gates, information can pass on the interconnect from one point on the plane to another point on the next plane. The schematic of a possible processor that employs symbolic substitution and gated interconnects in Figure 3.2.1 shows how stacks of logic gates and interconnect optics would be configured. It shows that the processor employs free space interconnects in three dimensions and at that only one plane does the pixel layout exactly follow the linear array of processors described above.

2.4 Scalability of SPARO

The problem of scalability is governed by the maximum feasible size of the planar optical array. We will be concerned mainly with the height of the array which is related to the size or the number of nodes in the CG. Considering the physical limits of resolution of the pixel in the optical array, our recommendations are that single array sizes be limited to 1000 x 1000. While this may not be large enough for accomodating programs written for large applications such as expert systems, we do have an approach for scaling the size of the architecture.

We propose to cascade multiple optical arrays in a circular configuration as shown in Figure 3.2.2. In this ring-like connection, the arrays are placed laterally end to end such that the complete configuration generates a large circular array. The interconnection network is placed on the inside perimeter of the ring. In this manner the effective size of the array can be increased by an order of magnitude or more to handle problems of larger size. Scaling the SPARO architecture in this manner would require increasing the size of the address fields. This can be easily accomplished by increasing the width of the optical array. No other changes would be necessary.

2.5 Overview summary

As it is proposed, SPARO will perform combinator graph reduction using an array of processors on which the combinator graph is directly mapped. The array of processors is expected to be constructed from different optical systems. Optical systems performing symbolic substitution will be used for implementing control and some of the logic operations, while optical registers will be used for movement of data. In the next section the concepts for the operation of SPARO will be described.

3. PRINCIPLES OF OPERATIONS

To handle the complexity of designing a computer with primitive optical components, we employ a hierarchy of levels of operations that corresponds to levels of functionality. This categorization is a framework applied to the development of the architecture to systematically reduce the complexity of the task. The categorization is similar to the one used in electronics. There the highest level of representation is the source program which is compiled into machine code. Machine code in turn (in some machines) is decomposed into register-transfer instructions representing the actual hardware operations of the processor. With SPARO, we expect more levels because the lowest levels of operations are even more primitive than those in electronics. The different levels are denoted as the levels of instructions:

1) source program level (e.g., LISP code),
2) SKI rules (combinator graph),
3) macro-instructions for combinators (S, K, etc.) and functions (+, *, etc.),
4) mini-instructions for realizing macro-instructions (e.g., for graph traversal),
5) micro-instructions - level intermediate between mini-instructions and symbolic substitution rules
6) symbolic substitution rules.

We will see later that not all levels are totally independent. It is also possibly that some levels may be virtual in the final design. (The machine will have no elements which correspond to operations at that level.) As we shall see later in Section 5, some operations, such as function evaluation, have to utilize the other two classes of operations. Although this classification mixes operations at different levels, it provides us a way to make tractable the difficult problem of designing a complex architecture. The complexity of design should be obvious when one notes the disparity in the level of functionality between the high-level (LISP) programming language operations and those of the symbolic substitution rules.

We describe the SPARO operation principle in the next three subsections. In the first subsection, we outline the control mechanisms or the mechanisms that have been used to implement the flow of control in the processor nodes. The second subsection provides a brief summary of the CGR evaluation strategy based on the normal order reduction introduced in Section 2. We emphasise the scope of parallelism in our evaluation strategy. The final subsection explains the CG representation in SPARO and how it relates to the evaluation strategy employed in SPARO.

3.1 Control Mechanisms

Implementing the SPARO control consists of two parts. The first is to determine the appropriate structure for the processors that realize the higher level functions for CGR. The second part is the implementation of the supporting interconnection network that both accesses indirect data and allows transfer of data from one location to another in the optical plane. The interconnection network is subservient to the control since any operation of the network is initiated only by a request from a node.

Sequencing of instructions to execute a complex operation is done quite differently than in electronic computers. In electronic computers the sequence of the instruction is either hardwired or microprogrammed. In either case the control unit

is centralized, i.e., separate hardware is dedicated for determining the control sequence. In SPARO, the processor executes using a technique we call 'instruction passing'. In instruction passing, the currently executing macro-instruction invokes the next function by sending a message to the appropriate destination node. The message is relayed by the network to the destination node which then executes the macro-instruction. This is analogous to passing the entry point of a process in a conventional multiprocessing environment. The control unit for SPARO is thus decentralized.

Besides avoiding the use of storage elements, this method of decentralized control can expose a high degree of parallelism because any number of nodes can be performing purely local operations, while the network passes many instructions, in parallel, from node to node.

SPARO does not have a separate memory management module since there is no separate memory. As is clear from our design approach, the control and memory do not have separate locii of operation. The control information has been integrated in the SS rules that also manage the required data movements, graph mutations, and function evaluation. This approach has been chosen to exploit the locality of SS operations.

3.2 Evaluation Strategy

The evaluation strategy corresponds to the normal order CGR [8] that was mentioned earlier. Normal order CGR allows the possibility of parallel reductions of reducible expressions, especially, strict functions [12]. Moreover, during the evaluation of a single redex, we attempt to exploit as much parallelism as possible in the execution of the fine-grained operations. Because different nodes can operate independently, such parallelism is possible even at the level of a single combinator reduction. To make such parallelism possible for correct operation, we must design each node as a finite state machine that can execute a specific set of instructions. The detailed design of the processor node is discussed in Section 5.

Examination of the combinator reduction algorithms reveal, that besides activating and suspending the nodes of the processor and initiating lateral moves, the processor generates a large number of network messages. Since many reducible expressions or redexes can be evaluated in parallel, there can be many possible node accesses occurring at the same time. This leads us to believe that the network may turn out to be a source of contention. The possibility of contention is not obvious unless a closer look is taken.

3.3 Graph Representation

The limited arity or out-degree of nodes in the CG makes adjacency list representations of the graph very attractive. In such a representation (Figure 3.1a), there are as many elements in the list as there are nodes in the graph. Each element in the list is comprised of a node and its children. To represent the CG, the list will have three fields: current node number, left child, and right child. Since the degree of each node is at most two, the complete graph can be represented in an array of bits consisting of three essential fields (Figure 3.1b). The first field maintains the node number or the address of the node. The remaining two fields indicate the node numbers of the left and right child and thus

act as pointers to child nodes. This array can be mapped, with minor modifications for CGR, into a two-dimensional optical array on which symbolic substitution is applied.

The adjacency list representation makes graph traversal from the parent to child nodes easy since information on the child nodes are included in the row for each node. Graph traversal from a node to its parents is more difficult since a node can have multiple parents. Information on the parent nodes is important since the result of a subgraph reduction is always sent to its parent node. The node representing the result of the reduction cannot be freed or deallocated from the array unless one can ensure that no other node is waiting on it. This observation implies that on completion of the reduction of a node (or subgraph), the control unit of the node must be aware of the number of parents that it possesses. To facilitate this, the parent information is included as fields of the node.

A CG node can possess any number of parents. However, in the SPARO node we have limited the number of parent fields to two. If more than two parent nodes are possible, they are connected by using a special '&' node that is used to string a list of parent nodes (Figure 3.2). Two parents fields are chosen for the following reasons. First, a single parent field cannot save an evaluation request from a second parent if the node is evaluating upon the request of the first. Second, if more than two parents are possible, the & node can be used effectively to store two parent fields at a time. When more than two evaluating requests arrive from the parent nodes, they can be saved in the & nodes.
While we have provided a broad overview of SPARO, a much more detailed account of the structure of SPARO is necessary to explain how graph reduction operations are executed.

4.1 MORPHOLOGY OF SPARO

While the control mechanism of SPARO is provided in the next section, the purpose of this section is to provide the necessary background, information on the node and message formats as well as some basic macro-instructions used by SPARO. We first describe the formats used for representing the data in processor nodes and messages. Certain mini-instructions are common to all macro-instructions that SPARO uses. These are described in the second part of this section.

4.1.1 Data Structures and Formats

There are two main data formats we are concerned about. The first is the format for the nodes in the CG. This corresponds to the data format of each row of the SPARO processor. The second data format is that of the message. The message format determines the format of the rows in the network and the network processor. While data formats specify the exact physical representation of the data, data structures indicate how, at a higher level, the data is structured or organized. The data structures to be considered are dictated by the high-level programming language used to specify the program. We will therefore consider the representations of the different data structures used in SPARO for CGR.

4.1.1 Data Structures and Data Types

As discussed in Section 2, the generic data structure for SPARO is the graph.

Lists, which form the core data structure in functional languages such as LISP, are just a special case of a graph. We will therefore only consider the representation of graphs. Details on graph representation were covered earlier in Section 3.3. We will mention that when a general list representation is sought, the linking of the elements of the list can be done by the LISP operator CONS or its combinator equivalent. Turner [8] suggests a combinator P as the curried version of the CONS or pairing operation. Thus a list of three elements a, b, and c represented in shorthand LISP notation as:

a:(b:(c:nil))

will be written as:

P a (P b (P c nil))

It is, however, more convenient to use CONS as a functional primitive rather than the combinator P. This would greatly simplify the control of the processor nodes.

Since details on other aspects of graph representation were provided earlier in Section 3.3, we list here only the possible data types that have to be distinguished during CGR. The use of the different data types will become clear in the context of how combinator reduction is done.

Due to the binary nature of the CG, a fixed size adjacency list representation is very attractive. For simplicity, we will discuss graphs with atomic node values. Complex data structures, such as lists, will be represented as a combinator expression of list atoms and the CONS operator. The data fields in a node of a CG are distinguished by the type of node it represents. Atomic values (including combinators and functions) are maintained only in leaf nodes whose right child always contains the value by convention. The & node, described earlier in Section 3.3, is used to string together multiple, specifically more than two, parents of an argument. A node that is neither a leaf node nor a & node is called a regular node. Thus, a node type is either & or regular or leaf.

The parent fields are always pointers and therefore do not require typing. However, presence bits are required to indicate the existence of a parent. Each node has two associated parent fields that are dynamically updated.

While we have described data types and data structure, we have not shown what formats are used for the nodes in the processor or for the messages in the network and the network processor. We describe these next.

4.1.2 Data Formats

Since each node in the graph is identified by its node number, searching for a node requires providing the network with the destination node number. So each path in the traversal of the graph requires determining the destination node and then providing the network with this information. When traveling down the graph, the left child (LC) or right child (RC) information is required at a node. For travelling to the root, the parent node information is required. Since data can be shared in the CG, a node can have multiple parents. Therefore, the parent value depends on the path by which the node is reached and is dynamically updated.

The layout of the node in the processor is shown in Figure 4.1. The required fields are:

Alloc bit (indicating whether the node is allocated or deallocated),

Activation bit (whether any operations are occurring in the node, indicates if SS rules should be applied in this row),

State of node (one of three values: unevaluated, evaluating, and evaluated),

Instruction bits (representing the macro-instruction currently in execution),

LC,

LC type (data type such as immediate, relation or function),

RC,

RC type,

Left Parent, and

Right Parent.

The network and the network processor contain specific registers or buffers. The registers are required to save incoming or outgoing message contents. The processor generates messages which are sent to the network via the network processor as shown in Figure 4.2. The format of the messages determine the structure of the network and the network processor. Only four fields are necessary for specifying a message:

instr or instruction that will be executed in the destination node),

dest or destination (node number),

source (current node number), and

two data fields data1 and data2.

The source field is usually the parent of the destination node.

4.2 Basic Control Instructions

A basic set of mini-instructions is used to write the macro-instructions for SPARO. These instructions are used to control the operations within a processor node, the sequencing of instructions, and the processor memory management. The instructions are described by their actual format.

4.2.1 Mini-Instructions

Three basic mini-instructions make up the macro-instructions. These instructions

move data between registers, initiate data transfers between processing nodes, and control basic and control macro- and mini-instruction execution. All of these are block structured and can take multiple arguments.

The simplest instruction is LOAD, which moves data from register to register within a node. Condition flags are also set with the LOAD instruction.

Conditional statements are represented by IF-THEN-ELSE structures, while CASE instructions are sometimes used to represent multiple IF-THEN-ELSE sequences.

The SEND instruction is used to construct messages in the processor to be sent by the network processor. It sets up the data in the output buffer (in the network processor). This buffer has four fields: dest, instr, data1, and data2. The network processor then sends a message containing these four fields and one indicating the current node (source).

4.2.2 Memory Management

The memory management operations are of two types: allocate and deallocate. An allocate operation is used to create and assign new nodes in the processor, while a deallocate instruction is used to free nodes that are not required anymore. The deallocate instruction is thus used to perform garbage collection. Since we are limited in the size of the array, memory management is an important function in the processor.

4.2.2.1 Allocate

This function uses the network to allocate a node in the graph. A message to a free node is sent using an indefinite form of the send instruction as shown below:

```
send
 out_buff/dest <- ?
    out_buff/instr <- instr
    out_buff/data1 <- data1
    out_buff/data2 <- data2
```

where '?' indicates any available node, and instr, data1, and data2 refer to an instruction and data to be passed to the newly allocated node.

The network then delivers this message to any node that does not have its 'alloc' bit set.

4.2.2.1 Deallocate Macro-Instruction

The complement of the allocate instructions is the deallocate (D) macro-instruction. It is more complicated since it must allow for the possiblity of two parents.

The basic operation of the D macro is to check if the node has two parents. If it does, then the node is marked as having only one parent. If the node only has one parent, then the node can be deallocated. Checking for multiple parents will be done explicitly within the macro-instruction that invokes the deallocate instruction.

5. CONTROL IMPLEMENTATION IN SPARO

In this section we describe how the control functions in SPARO are implemented. We focus on the macro-level architecture and list the algorithms required for the basic set of functions and combinators. The basic set of functions is that which is necessary for executing a pure functional language using graph reduction.

The set of combinators chosen is S, K, I, B, and C. Although S, K, and I constitute the basic set of combinators, B and C will be required for optimizing the size of compiled code. We have assumed that the CONS operation is implemented as a primitive function rather than use the P combinator. The fixed-point combinator Y will not be used to represent reentrant recursive code. Instead, compiler annotations will be used in the compiled graph for purposes of efficient graph reduction in SPARO. Compiler annotations will also be used to distinguish reentrant iterative code.

When considering the list of primitive functions we will use LISP primitives although the macro-level algorithms can be adapted easily for any pure functional language such as SASL. The primitive functions considered will include the primitive 'eq' for comparison and 'cond' for conditional testing. The arithmetic primitives will consist of the unary minus (-) and the binary plus (+) operators. Only one Boolean primitive NOR (or NAND) will be considered. Separate input/output functions IN and OUT will be assumed to read and write data to and from SPARO. The input and output primitives will not be considered here.

This section is divided in three parts. In the first subsection we describe the abstract finite state machine (FSM) of a processor node to explain the states of a node. The second subsection outlines the macro-level algorithms for both combinators and functions. In the third subsection we show how higher level control structures, such as iterative and recursive structures, can be implemented efficiently in SPARO.

5.1 Abstract Finite State Machine Model of a Processor Node

Although each processor node is an FSM that executes the different control sequences for reducing combinators and functions, it can be considered as an FSM at a higher level. This level of abstraction corresponds to specifying the state of evaluation of the node. Specifically, a processor node can be viewed to be in one of three external states: not evaluated (nev), evaluating (evg), and evaluated (evd). Initially, all nodes are in the nev state. When the graph embedded in SPARO is to be reduced, the root (user) sends an evaluation request. This request is manifested as an 'apply' instruction from the root node and is sent to its leftmost descendant as prescribed by normal order graph reduction. The state transition of a node is determined by the external state information (nev, evg, or evd), the type of its LC, and the instruction it receives as input. Thus, the actual state of a node is determined by its external state and the type of its LC while the input to FSM is the instruction received. We discuss below the state transitions that can occur.

If a node does not specify a combinator or a function then the following actions are taken. A message is first sent to its LC to transmit the apply instruction. The current node is then placed into the evg state. If the node is a combinator or function node, then the associated macro-level sequence is invoked. In case of a

function evaluation the collection of all required arguments is initiated.

A transition from the evg state to the evd state is made if the node is reduced to a value. A node can be reduced to a value only as a result of a K or I reduction or by a function evaluation. After the reduction a node sends an 'evaluation complete' message to its parent. An evaluated node can be deallocated only if no other parent is waiting on its value. Note that when multiple apply requests arrive at a node with multiple parents, the requests are saved in the parent & nodes. Each & node can hold two apply or evaluation requests. The parent and & nodes are set into the evg state. When the child node is finally evaluated, the 'evaluation complete' message is sent to all waiting parent nodes that had requested the evaluation.

Note that since subgraphs are shared and not replicated, as in string reduction, whenever an evaluation is requested of a shared subgraph, new nodes must be created to represent the mutated subgraph. Thus, nodes have to be allocated during a reduction. As will be seen in the detailed macro-algorithms, the complexity of SPARO is not in the actual reduction sequence but in the management of the shared subgraphs.

An explaination of the control algorithms that define the specification of a processor node in SPARO are given next.

5.2 Combinator Reduction and Function Evaluation

The macro-instruction sequences or macros for the basic set of combinators and primitives are given in the Appendix. All sequences are written using the mini-instructions introduced in the previous section. The macro-instructions for combinators and functions are written in specific format so as to expose the available parallelism in each step. The format used is described below.

Every macro-instruction description is preceded by the state the node is in and the instruction that it receives in the message from the network. This specifies the current state and the received input of the node. A state transition can occur only on the receipt of a message. The node undergoes a state transition, whereby certain actions are initiated and messages are sent out on the network. These actions correspond to the output of the FSM of the node, and are dictated by the instruction specified in the message. Actions consist of assigning values to the fields internal to the node and the fields of the outgoing message buffer (in the network processor). The incoming and outgoing message fields are distinguished by the use of prefixes in buff (buffer) and out_buff, respectively. Messages are invoked by the 'send' mini-instruction.

The field assignments in a node can be done with parallel register to register moves or loads (denoted by <-) where the actual writing of values can be done at the end of the move cycle. This permits a swap of the form: a <- b, b <- a to be done in one cycle. The convention adopted here, however, is that the internal fields of the the node are assigned first (possibly in parallel). The fields of the outgoing message are then assigned (loaded) with the proper values (all in parallel).

The three external states clearly do not suffice to completely describe the FSM that represents the processor node. We will therefore use two state representations: an

external and an internal state. The external state refers to the state of the abstract FSM, and can have one of three possible values (nev, evg, or evd). The internal state of the node is defined by the macro- and micro-instruction being executed in the node and by the values of certain control fields and flags. As far as the input to the FSM is concerned, it is only the instruction field of the message that initiates the actions to be taken. During a specific macro-instruction, the actions taken are governed by the node type (i.e., regular, &, or leaf) of any of the LC, RC, or the data fields of the incoming message.

The macros described later are written as a sequence of actions that take place during a state transition. Intermediate states of a node are indicated implicitly by the specific instruction being executed, and the values of the control flags and fields. Specific instructions being executed in the node are indicated by setting specific bits associated with those instructions. Thus an S1 micro-instruction, the first instruction executed after an S combinator is recognized, is indicated by setting the S1 bit in the instruction field of the node. Certain control fields and flags are used for purposes of control in a SPARO node.

5.2.1 Control fields and flags

Our preliminary analysis of combinator reduction and functional evaluation algorithms reveal that five Boolean flags are necessary for use by the macros. These are:

2_parents_present -- This flag indicates that both parent fields of the node are occupied, or that the node has two or more parents,

2nd_apply_received -- This flag indicates whether the second parent of a node has requested an evaluation; applies only to nodes for which 2_parents_present is true,

doing_S -- This flag indicates that at the top level (root node of a combinator subgraph) an S rather than a B combinator is being executed,

awaiting_arity_incr_replies -- This flag indicates that two new parents have been added to the parent list of a shared argument node (in case of a S, B, and C combinator reduction),

awaiting_eval_completes -- This flag indicates that both arguments of a binary function have been evaluated; the binary function can now be applied.

Besides these flags, two other fields are used for describing a node type and the level of the combinator subgraph. These are:

node_type: regular, &, and leaf, and

combinator_level: 1, 2, or 3, where the number refers to level in the combinator subgraph; 1 is the level of a node whose LC is a combinator (S, B, or C), while 3 is the root of the combinator subgraph.

We have assumed leaf nodes for atomic values in the CG, i.e., atomic values have independent existence. Thus all arguments of a combinator or function are distinct nodes. While this results in inefficient use of storage, it implies a more efficient

control. Maintaining separate identities for atomic values implies that every argument is a pointer to a value. Such a representation eliminates the need for checking whether every argument is a pointer or value, and considerably simplifies the macro-instructions.

Besides the aforementioned flags, an 'annot' flag is used to indicate that a node is annotated and cannot be reduced on first evaluation. Annot nodes are used to describe recursive code, and are dealt in detail in the next subsection.

The memory management in SPARO is directly built into the control algorithms. Thus, the macros explicitly control deallocation and allocation of node. A separate state bit 'dealloc' is assumed to denote if the node is free or in use. An important fact to note is that a shared subgraph, either appearing as a LC or a RC of a node, that is not reducible in itself, cannot be deallocated. On receiving the first request for evaluation, new nodes must be allocated to represent the mutated or reduced subgraph. The shared subgraph will remain intact until the last evaluation request. Only then can it be deallocated.

The communication between the processor array (nodes) and the network and the network processor is handled by messages. Within a macro, a message (send) is specified by four fields: dest, instr, and data1 and data2. The data2 field is not always necessary. The source field does not need to be specified since a message initiated in any node will always use the current node address as the source.

Functional units that execute primitives, such as + and AND, are assumed to be located by a message with the destination address set to the name of the function (denoted by functional unit). This assumption makes the implementation of the control unit of SPARO independent of the implementation of the functional units. The assumption also implies a pessimistic implementation where a few (or, in the worst case, only one set of) functional units are shared by many nodes of SPARO. Ideally, each node should have its own set of functional units so that there may be no contention in access of the units or the network. Physical and technological limitations, however, may prevent realizing the ideal situation.

While the macro-algorithms have been listed in the Appendix, we describe here an example sequence of macro-instructions that would be used to reduce an S combinator. Refer to the Appendix for the detailed macro-instruction.

5.2.2 An example combinator reduction

The combinator expression is assumed to be simply (S f g x). The CG for this expression is shown in Figure 5.1. Although sharing of the top three nodes (1 - 3) is not shown explicitly, the implications of doing so are discussed below.

When the CG is unevaluated, all four nodes of the CG are in the nev or unevaluated state. The reduction is initiated when an apply instruction is received at node 3. The parent information is then stored in the P1 field so that the final reduced value of this CG can be communicated back. Since node 3 is a regular node it sends an apply to the LC or node 2 and sets node 3 in the evg or evaluating state.

Nodes 2 and 1 go through the same sequence of actions as node 3 since they are

also regular nodes. The apply instruction thus arrives at node 0 which is a leaf node. Thus, the message msg_normal is constructed and sent up to node 1 (sendup1). Since node 0 is a leaf node, it is not expected to be shared and is therefore deallocated. The message sent to node 1, which was in the evg state, contains the instruction eval_complete and the S atom in the data1 field. The macro-instruction corresponding to S is thus invoked. Since the buffer data in the message is in the combinator/operator set (CombOp), the LC is loaded with the atom S, and the macro-instruction comb1 is invoked.

In comb1, the combinator_level is set to 1, and the message msg_mut_arg1 is constructed. Since an S combinator has been recognized, the message will specify S1 as the next instruction. In comb1 a check is done to see if node 1 is shared. If node 1 has two parents, indicated by the flag 2_parents_present, then the argument f cannot be consumed by this reduction and a pointer to f must be created. This is achieved by the macro-instruction arity_incr, which increases the number of parent fields of the argument f. The arity increase procedure is simple if f had only one parent, otherwise an & node has to be allocated using the macro-instruction and_node_alloc. In the latter case the macro and_node_alloc sends back the address of the newly allocated & node (which now points to f) to node 1. Node 1 then has the new & node as its RC. If however f is not shared, then node 1 is deallocated, else the node remains in the evg state.

The message msg_mut_arg1 from node 1 arrives at node 2 bearing the instruction S1 and the argument f (actually the pointer to f). The combinator level is set to 2. The combinator level number determines the correct response on completion of the arity_incr macro. Since node 2 was in the evg state, the LC is loaded with the argument f, and comb2 is invoked. In comb2, the message msg_mut_arg2 is constructed. Msg_mut_arg2 specifies the instruction S2 for the node 3. As in node 1, node 2 can be shared. In such a case the parent list of both f and g have to be updated. Thus, as in node 1, arity increase messages are sent out for each argument. The Boolean awaiting_arity_increase_replies is used to indicate that the new parent fields has been added. As in node 1, node 2 can be deallocated if it is not shared, or else it remains in the evg state.

The message from node 2 arrives at node 1 instructing that S2 be executed. The combinator level is set to 3, and the doing_S bit is set to indicate that S combinator reduction is in process. As in the other two nodes, the argument x may be shared so the arity_incr macro is invoked to ensure proper sharing. When the control returns to node 3 (in arity_incr), it sends out a request (S3) for the creation of a new node 4 for the expression (g x). S3 then sends back a message to node 3 with the address of node 4, and specifying the instruction S3_reply. Node 3 then sets node 4 as its RC, and sends out a request (S3_with_apply) for the creation of a new node 5 for the expression (f x). S3_with_apply starts evaluating (f x) in normal order. When node 5 is evaluated the control is sent to node 3, which is set as its parent. The function or combinator subgraph resulting from the evaluation of node 5 then uses node 4 as its argument.

While simple control structures involving combinators and functions have been addressed, complex control structures such as recursion, common in functional languages, have not been discussed thus far. Handling such control structures in SPARO is discussed next.

5.3 Handling Complex Control Structures

We describe here how the control in the processor can be designed to execute recursive and iterative structures. Recursive subgraphs are discussed in more detail since the method for handling iterative subgraphs is just a special case of the former.

5.3.1 Recursion

The typical method for executing recursive code is to expand out the recursive call until closure is reached. The control flow sequence is usually maintained by using a stack. Since stacks and other explicit memory structures are difficult to implement in optics, we have chosen an alternate route. Instead of executing shared code and saving different contexts on stack, we copy the subgraph for the recursive structure and evaluate or reduce each instance separately. Thus each instance of the recursive code is explicitly generated and reduced during CGR. There are two implications to this approach. First, the recursive subgraph has to be copied each time. Second, the original subgraph cannot be reduced and is thereby destroyed before closure is reached. These two issues are examined next.

Copying a complete subgraph at runtime will be very expensive since the complete subgraph is first copied and then executed. However, such a 'copy and reduce' technique may not be necessary. One possibility is to pipeline the copying and reduction operations. Thus, while the original graph is being copied (in normal order), the apply instruction can be sent down the incomplete graph in normal order. Since copying requires allocating free nodes and sending data via messages, the speed of copying will depend on the speed of the network. To prevent evaluation of an incompletely copied node, certain control constructs will be required to suspend evaluation until its child nodes have been specified.

Normal CGR reduces and thus destroys the original CG. That is, the combinator graph is not reentrant. In the electronic case the compiled CG is stored in some main memory, and recursive routines share the same code maintaining different contexts at runtime on the stack. Since we do not have separate memory, we must make other provisions for maintaining the recursive subgraph. To avoid use of a separate special memory, we ensure that the original recursive graph is not destroyed until closure is reached. To distinguish the recursive portions of the code from the rest during execution, we use compiler annotations for all nodes that constitute the recursive subgraph. An annotated node will have its 'annot' bit set. We now describe the control sequence that occurs when an annotated subgraph receives a DF instruction. Note that another option to the annotated node technique would have been to distinguish between compile-time generated nodes and run-time generated nodes where the compile-time generated nodes are never destroyed. Conceptually, there is no difference between the two options, and so we have decided to examine the annotated case.

When the root node of a recursive subgraph receives an instruction to evaluate (apply), it initiates a copying algorithm. If an incremental copying algorithm is used, i.e., copying and graph reduction are pipelined, it can be executed recursively. A copy instruction does the following: the present node is first copied in a newly allocated node (the allocation is also initiated by the copy instruction), and a copy instruction is sent to its LC and RC. Each child node then again

invokes the copying algorithm. The copying stops when leaf nodes are reached. After the first node has been copied, an apply instruction is sent to the first copy of the recursive subgraph. In this manner the copying and evaluation of the subgraph can be accomplished in a pipelined fashion.

The correct sequencing of the recursive evaluation is specified by the linking of the copies of the subgraph (shown in Figure 5.2). Let the head or root node of the original recursive subgraph be called H. The node that specifies the recursive call, i.e., the node in the body of the subgraph that points back to H, will be denoted by T for tail. These nodes in the copies will be called H_1, T_1, H_2, T_2, etc.. The correct sequencing for evaluating the recursion proceeds as follows. The parent node P of {H,.., T} sends down a request for evaluation, an apply instruction, to H. Since H is the first annotated node, it initiates the first copying sequence. The copy consists of the sequence $\{H_1, .., T_1\}$. The tail T_1 is set to point to the original head node H. H sends an evaluation request to H_1 with the address of its parent P. Since the subgraph $\{H_1, .., T_1\}$ is not annotated, it will be reduced. If the recursive condition (the condition that determines the recursive call) is satisfied, T_1 is evaluated. Since T_1 points to H, another copy of the original subgraph is made with the new tail T_2 pointing to H. H_2 is sent an evaluation request from H with the address of T_1 since T_1 is the new parent of H. In this manner the subgraph copies are linked to emulate a nested call structure.

The recursive spawning of new copies continues till closure is reached when the recursive condition is false, say at the kth copy. At this stage T_k is sent a deallocate message since the recursive condition is false. Since T_k points to the original subgraph, {H,.., T} is recursively deallocated. In parallel the kth copy reduces and sends back a completion message to H_k, which in turn sends a completion to T_{k-1}, the tail of the (k - 1)th copy. The (k - 1)th copy of the subgraph is then reduced. The reduction proceeds until the first copy is reached. Since H_1 has P as its parent, the final completion message is sent from H_1 to the original requesting node.

It would appear that handling recursive calls is more efficient if block copying rather than incremental copying is used. This does not affect the control sequence described above. In block copying, a complete block of nodes that constitute the recursive graph can be copied in a single step. This avoids the problems of flooding the network with messages to allocate nodes and checking if child nodes have been allocated. Furthermore, all nodes would be local to the block thus reducing contention in other parts of the processor. Block copying can only be feasible if the size of the recursive subgraph is limited to a certain size.

5.3.2 Iteration

The problem of maintaining the iterative subgraph until iteration is complete is similar to that of the recursive case. Therefore, a similar solution using compiler annotations is chosen. The only difference is that unlike in recursion the life of the iterative structure is usually known at compile time. Furthermore, the different iterations do not exist at the same time. Only the current iteration must be saved. The copying of the original subgraph is initiated if the termination condition is not satisfied. A reduction can proceed as soon as the copying has completed or is in progress in the case of incremental copying. Each iteration is completely reduced before the next iteration is generated since no nesting, as in recursion, occurs. We

do not emphasise iteration in our control implementation since most procedures in LISP are written using recursion. We note, however, that in terms of space constraints, recursion is more demanding because multiple copies of the subgraph exist simultaneously.

Having examined the detailed evaluation strategy for CGR in SPARO, we now examine the optical implementation of the different components of the architecture in the next section.

6. OPTICAL IMPLEMENTATION

The execution of the instructions described in the preceding section assumes that each node contains some processing capabilites and the ability to send messages to other nodes. The discussion also alludes to the use of a portion of the processor that arbitrates data transfer to and from the network. This section describes possible implementations of the each of these component systems. We first describe the basic optical components assumed available and employed in SPARO.

6.1 Optical Primitives

Only a few optical elements are required to implement the structures needed for SPARO. It is assumed that suitable nonlinear optical gates can be found to perform the logic operations required. Interconnects between logic planes will likely employ classical optical components such as mirrors, beam splitters, and lenses as well as holographic deflectors.

The two techniques that SPARO employs are symbolic substitution and gateable interconnects. Symbolic substitution is a technique to perform complex logical operations by the manipulation of patterns. Its main attraction is that it is well-suited to operation in parallel and it has a relatively straightforward implementation in optics. Gateable interconnects are nothing more than masks that allow light to be transferred to specific locations in the next logic plane in the system. In SPARO, symbolic substitution is employed to perform the control operations within a node, and gateable interconnects are used to transfer information between fields.

An optical system that performs one symbolic substitution rule consists of two portions: the recognition optics and the scribing optics. In the recognition optics, the fields where symbolic substitution are to be performed are optically split into several copies. These copies are then shifted in varying directions and distance and imaged onto a plane of optical gates operated as thresholding elements. In this manner, specific patterns can be recognized. The light out of the optical gates denotes locations where a pattern was recognized. In the scribing optics, this output light is optically split into several portions, then shifted and recombined creating the desired output pattern.

Each rule in a symbolic substitution system requires a specific recognition and scribing optics for that rule. It may be possible, however, to combine shifted images, or partial results of recognition and scribing to reduce the complexity of many rule symbolic substitution systems.

Gateable interconnects could be realized by using a pixel from a control field to control several optical gates configured as a programmable mask. If the light from

the control pixel is present then light would be passed through the mask. Gateable interconnects could be realized by using such programmable masks in the paths of light connecting the output from data fields to the imput of other data fields. Figure 6.1 shows how symbolic substitution (SS) and gateable interconnects could be used to move data around in an optical system. Specific pixels in the region where symbolic substitution is performed are used to control masks that send information from register A to registers B and C.

6.2 Optical Layout

As described earlier, the optical system is laid out as a linear array of nodes. The elements of this plane are optical gates that provide the nonlinear functions required for symbolic substitution. Free-space optical interconnects provide the connections between fields in the nodes and on the network as well as for symbolic substitution. Connections are provided for all processing elements in parallel to facilitate simultaneous operation of all of the processors. The linear layout is required to reduce the number of possible interconnects between pixels on the plane so that diffraction effects can be managed.

Figure 6.2 shows schematically how the plane is functionally split into three sections and how that plane makes up part of a larger system. The block in the lower portion of the figure represents the optics employed for the interconnects between planes of nonlinear elements. Logical operations are performed by interconnecting the outputs of optical gates from one plane to the inputs of gates in the next plane. The blocks and planes in Figure 6.2 correspond to the blocks and planes in Figure 2.1. A complete cycle of light around the loop shown in Figure 2.1 represents one machine cycle. The various stages are required to perform the operations needed by symbolic substitution and to set up the transfers between registers.

To explain the macro-architecture we will examine the interaction between the processor portion of the optical plane with the network portion. An intermediate stage is necessary to handle the management of data transfer between the network and the processor. We will refer to this stage as the network processor. It will be seen that the network processor stage alleviates the network contention problem since, we believe, that a major portion of the execution time will be expended in moving data from one node to another. Since there is ample scope for parallel data movement, there is a high probability of contention for access to the network.

6.3 The Architecture of the Network

The network used by SPARO is best described as a micro-area network. It has an extremely simple protocol, but it can pass messages in parallel between the many nodes of SPARO. As was evident from the macro-level descriptions of the SKI combinators, it is used by the processor by invoking send instructions. Here we examine one possible configuration of the network. It was not designed for high performance, but rather to demonstrate that a communications channel could be developed which was in keeping with the overall system requirements.

The requirements used to develop this network, and which must be met by any alternative, are:

parallel or near-parallel transmission of many messages,

simple (from the node processor point of view) contention management,

simple access,

supports node allocation,

nonblocking (partially managed through processor design),

The functionality of the network is determined by the nature of operations required for CGR. The gross functions that are required as capabilities of the network are graph traversal and data moves. These in turn require that the network must be able to locate nodes in the array by their node numbers, and be able to read/write data in and out of the data fields.

Another required feature of the network is the parallel access of nodes in the plane. This is critical since we want to exploit any parallelism available during CGR. When different subgraphs can be reduced simultaneously, the network should be able to accomodate the data moves and traversal in disjoint segments of the graph. Efficient parallel access of the nodes in the graph implies that there should be no contention in the use of the network. As we will see later, addressing contention will strongly influence the design of the architecture.

The obvious approach for the network to access nodes in SPARO is binary decoding of the addresses or the node numbers. However, previous related work on memory design in optics has indicated that binary decoding in optics is inefficient both in space and time [1]. Our approach has been to sequentially traverse nodes and check node numbers (which are ordered in the array) whenever an address is given. A number of nodes can be accessed simultaneously since all sequential traversals in the array occur at the same speed.

The basic operation of the network is to sequentially step a message from node to node along the linear array. If the destination matches the neighboring node, the data is transferred from the network to the network processor.

We will assume, in our implementation, that the network searches for the node number sequentially in the proper direction (up or down) of the array. The sequential search means that at every node a comparison is made between the destination node number and the current node number. While this appears slow, the advantage is that both the processor and the network (and also network processor which manages contention) can operate at the same speed and with the same cycle time. Searches for nodes can be done in either direction to allow for parallel searches crossing the same node without contention. Such a bidirectional search is implemented by providing two sets of fields for each node in the network. Figure 6.3 shows the format of the network for implementing bidirectional searches.

The network can be viewed as a sequence of registers, one set of which transfers data up the array while the other set transfers data down. In each register set the basic operation is the transfer of one register to the next as in Figure 6.3. On each processor cycle register data is moved up (or down) one step. Data transfer is initiated by placing a message on the network. To place a message on the

network, the processor first determines the relative location (up or down) of the destination. It then checks to be sure that on the next machine cycle its network register will be free by examining an 'occupy' pixel or bit. If it will be, the processor transfers the message to the network. If not, it checks again for a freed register.

On every cycle the network compares the message destination to the current position of the message. This is accomplished by comparing the node number field to the destination number field of the message. If the message is at its destination, it is moved off the network into the network processor. Otherwise, it is passed to the next register in sequence.

As described, contention for the network is managed by refusing to send a message if the network is busy. This method of managing contention reduces the buffering requirements of the system since the node can be made to buffer the information while it is waiting to send it out. Further, this method does not introduce more inefficiencies in execution since a node, in most cases, does no processing until its subgraphs have been reduced. This means that it can idle while waiting for access to the network.

Since multiple processes can be occurring on the same graph, a message may arrive at a node that is still processing. In such a case the message must be buffered until the current processing is complete. This buffering is provided by the network processor. The network processor manages two buffers for messages from the network. It only needs two buffers since our implementation of CGR has at most two parents and children. Such limited connectivity constrains the number of messages that can arrive simultaneously at a node, and thus limits the number of requests that may have to be buffered. The use of & nodes provides the extra buffering required in the case where traditional CGR would set up more than two parents to a node.

Note that two accesses proceeding from different directions can access the same node. This means that the network processor must contain two fields to store messages from each direction. This is the most storage required since our implementaion of CGR eliminates the possiblity of more than two different messages reaching the processor at the same time.

If the message data cannot be loaded into the node, it is maintained in buffers. During data moves from the network onto the processor, another portion of the network processor may buffer the message so that the instruction currently executing can finish.

The access time for the above interconnection network will be O(n) where the number of nodes in the array is n. Since all data access is not guaranteed to exhibit locality, the network may cause performance degradation. For this reason we will augment the network with extra connections such that it can bypass blocks of nodes. For example, if the network provides a connection between every node and its kth neighbor besides connecting adjacent nodes, the maximum access time of a node is $O(n/k + k/2)$. We refer to this augmented network as the 'supernetwork.' The best access time for the supernetwork is $O(\sqrt{n})$.

6.4 Architecture of the Processor

A study of the functionality of the processor shows that the control operations are relatively simple and few, especially at the macro-level. The aim for the development of an architecture for the processor is to determine the minimal set of instructions that need to be implemented.

The registers and functionality of the processor, described in Sections 4 and 5, require that the processors store information and transfer it between fields; the same functionality is required of the network. However, in the case of the processor, the control for data transfers is provided by the executing instructions.

Currently we have not completed the design of the processor architecture, but have schematically developed how it will operate. Figure 6.4 shows, schematically, the basic functionality of our concept for the processor. Bit fields in the instruction register control gateable interconnects between different lateral moves. The bit fields in the instruction register, in turn, are controled by symbolic substitution rules. We envision that the symbolic substitution rules would be developed from the mini-instructions in a two-step process (mini-instructions to micro-instructions to SS rules) so that the number of SS rules can be minimized.

At present only the functionality required for combinator graph reduction has been specified for the processor. Functional units that perform operations such as additions and comparisons will have to be added to complete the processor.

6.5 Network Processor

The network processor is a unit wthat buffers the request to and from the network. It computes the direction in which to send a message and assures that the message is saved in an empty register slot. A message is buffered in the network processor if it is received while the node is in an evaluating (evg) state. The message remains buffered until the current evaluation is complete.

While the use of the network processor combined with the limited number of parents and children solves the contention problem for high levels of parallel data movement, we have yet to address the lateral moves required in SPARO. The possible lateral data moves are:

network to the network processor fields,

network processor to processor fields,

processor to the network processor fields, and

network processor to network fields.

For implementation, one would select a minimal number of possible data field moves in each of these three categories.

Note that because the network processor works in concert with the processor and the network, it will require its own set of "instruction" registers which would keep track of its state relative to that of the processssor and the network. These instructions will control and sequence the moves to the network and the processor.

Again, as with the processor, symbolic substitution will be employed to set up the sequencing and gated interconnects will provide the actual data movements.

7. FUTURE WORK

We have presented a portion of an architecture to perform combinator graph reduction, which was developed to exploit the parallelism of optics. It was developed by examining the overall requirements of combinator graph reduction so that the resulting architecture can be scaled to large systems. To complete the architecture and make it viable for implementation several other aspects must be considered. This section describes possible upgrades to the connection network, addition of functional units, and extensions of the architecture to design expert systems.

7.1 Upgrading the Network

The network design presented in Section 6 could become a bottleneck if SPARO were to be constructed since a disproportionate amount of time is spent on communication. Further work on SPARO is necessary to develop a more efficient communications system. We describe several options in this subsection.

The linear network could be enhanced by the addition of the supernetwork mentioned earlier. This addition would transfer data across many nodes in one cycle rather than just one. Figure 7.1 shows how the supernetwork would be added to the single-step network. In the example shown, data could be either passed to the next node or it could be passed to a node 10 steps away. The optical implementation of this supernetwork is not expected to be much more difficult than for the single-step network; it only requires shifting an image by 10 nodes rather than one.

One problem in developing a network such as this is contention. Since data could travel on many paths from source to destination node, an efficient routing strategy would have to be developed. The increase in the number of data paths would also require a strategy to ensure that data could pass on the different paths without requiring buffering on the network.

Another related problem is node allocation. The scheme for memory allocation described in Section 4 would have to be modified, perhaps totally redesigned, to account for the multiplicity of data paths. It may even require the use of an allocation scheme as complicated as that used by the Connection Machine [9].

If a more complex routing scheme and memory management algorithm are required for the supernetwork concept to be effective, it may be fruitful to examine more richly interconnected topologies such as multistage interconnection networks (MINs) or hypercube connection schemes. While the average data access times in either topology is much smaller (O($log sub 2$ n) versus O(n)), there is an increased overhead in the processing of the routing and contention management algorithms. However, for large networks, it is possible that the richer interconnection topologies may be more attractive.

Functional Units

One of the components currently missing from SPARO is the set of functional units. For SPARO to be complete it must be able to perform more than just the functions required for combinator graph reduction. It needs to have primitive functions (such as +, -etc) included in it. Two approaches exist for completing SPARO. The first would be to add a functional unit to each node. The second would be to have special nodes attached to the network for the sole purpose of executing functions. We describe the tradeoffs for each approach.

Assigning a functional unit for each processor would provide the best performance. The ability to perform primitive functions without a large number of data movements would reduce the time taken by the functions. However, since most nodes do not, and may never do in the course of program execution, perform the primitive functions, the addition of functional units is very expensive in terms of unused "computational power." The addition of functional units to each node would require a larger optical system and a correspondingly longer cycle time. This waste of computational power and longer cycle time may be justified if it leads to better system performance than its alternative.

The second choice for implementation of functional units would be to use special nodes on the network for performing the primitive functions. This, naturally, would increase the traffic on the network and increase the time required for function evaluation. However, this approach would simplify the nodes and would not add a large amount of seldom-used hardware to the system. The contention of many requests for a few resources would then have to be addressed. An example of a potential problem would be more requests for addition than the number of functional units. Some requests would naturally have to be buffered, but the location and mechanisms need to be defined with careful regard to optics capabilities and limitations.

7.3 Extensions to SPARO for Expert System Execution

The current SPARO design is geared towards executing functional languages, specifically LISP. The design is also based on the execution of functional primitives alone. However, since our original goal was the design of optical processor for expert system execution, we will focus on how SPARO is extended, augmented, or modified for this purpose. In this subsection, we present an overview of expert system architectures and show where SPARO fits in.

A general expert system is an inference engine that operates on a knowledge base (KB), which is a knowledge repository for a specific domain. The KB can be in the form of rules, frames, logic, semantic networks, or procedures, although rule-based and frame-based expert systems are the most popular. We will therefore discuss only the rule-based and frame-based systems.

In a typical execution step in the expert system, inference rules from the KB are applied on facts or assertions that represent the real world. The body of facts is also called the working memory (WM). In the case of rule-based systems, (where most of the knowledge is conveniently expressed as experiential heuristics) the inference mechanism corresponds to the following steps. First, the KB is searched for rules that apply. This is the match or selection phase. Since multiple rules may be applicable, a conflict resolution, usually a prioritization, is done to choose one rule to fire. The chosen rule is executed, that is, the actions or conclusions

recommended by the rule are incorporated in the body of facts by addition, modification, or deletion. If the KB is composed of frames (if complex structural descriptions are necessary to describe the domain), similar steps are followed. The difference is in the searching mechanism of the frames that apply. Facts in the WM and the frames of the KB are usually organized hierarchically as opposed to a large sequential list of production rules.

There are two ways of viewing the architecture of the inference engine that comprises the expert system. One view is to treat the expert system as a loosely coupled system consisting of a hardware platform executing the application program, referred to as the execution engine for convenience, and the KB and its management, referred to as the match engine. In the second view the application program is tightly coupled with the KB. The difference between the views is mostly conceptual but in terms of efficient implementation this difference may be significant. In the first case, one could implement the match engine and execution engine separately, and thereby partition the functionality of the inference engine. In the second case, one has to incorporate some of the functionality of the KB management into the execution engine.

When considering an optical implementation of expert systems, we treat the two views as two possible options. In the first option the inference engine consists of SPARO and a separate match engine, while in the second option the inference engine is an extended version of SPARO.

The advantage of designing the match engine is obvious. Parallel associative optical memories can be used to implement the match phase. This would speed up the match phase, a typical bottleneck in rule-based expert systems. Such an implementation would also provide a significant advantage over electronic match engines because of the possible parallelism. In the second type of implementation, where the match is not implemented separately and the KB is part of the SPARO optical plane, associative searches cannot be used as effectively. The second type of implementation appears more appropriate for highly structured KBs such as those using frames where associative searches are not usually carried out. One must therefore conclude that a separate execution-match implementation is better for rule-based KBs, while the single inference engine model appears better for highly structured KBs.
The two approaches to expert system architectures, however, are not as different as it may seem; there are cases where the first implementation may be used for non-rule-based KBs. The degree of parallelism available during the searching of data in the KB depends on the structure of the data in the KB. Rule-based systems are not the only ones that allow for parallel searches of their KBs. In practice [13], frame-based systems, such as the KEE commercial expert-system building tool, allow production rules to be stored within and activated from frames to do inference. Similarly, rule-based systems commonly incorporate frame-like structures to facilitate the representation of large amounts of factual information. Thus an associative search of the KB may be required in either type of expert system. Furthermore, if frames are structured in a hierarchy based on a certain categorization, then each category could be searched associatively for the appropriate category. Such a nested associative search can still speed up the access of frames in a KB. In short, there is ample scope for using the executiom-match implementation even for non-rule-based expert systems.

While a separate match and execution units in the inference engines exposes parallelism during the match phase of execution, such an architecture has certain drawbacks. Inferencing in an expert system can use either forward chaining or backward chaining. In forward chaining, the firing of a rule (the discussion will be limited to rule-based KBs) will change the state of the WM and cause other rules to fire, and so on. In backward chaining, the goal to be met is translated into subgoals to be met, and so on, as specified by the rules. A rule is selected for firing on the basis of its consequent (conclusion or action dictated by the rule) rather than on the basis of its antecedent (conditions of the rule). Since a single rule is usually selected for firing at each stage, a failure to satisfy the goal would require backtracking to select a different sequence of rules. Backtracking that requires some form of stack manipulation is difficult in optics as seen earlier in the context of choosing the symbolic processing language. It would thus appear that optical symbolic processors are better suited for forward-chaining expert systems. Since diagnostic-type problems typically use backward chaining, an expert system based on forward chaining inferencing alone will not be as efficient in those domains.

7.4 Execution-Match Inference Engine in Optics

In this subsection we present our architectural view of the expert system that is based on our earlier work on SPARO. Given how cumbersome it is to build complex control structures in optics, it is desirable to keep the match as simple as possible. This implies putting as little functionality as possible in the match stage, and as much control as possible in the execution stage. A good example of such control complexity is the conflict resolution stage required in rule-based systems to select the rule to fire when many rules are possible candidates.

To simplify the match engine architecture we will consider the following structure. Consider the match to occur on a plane. If rules make up the KB, then each row on the match optical plane will be assumed to be on one production rule. Each row is then divided in two parts: the antecedent or the condition that asserts the rule, and the consequent that describes the conclusions or actions that follows. The antecedent or the consequent is expressed as a conjunction of clauses. These clauses could be written as a disjunction of other clauses.

A number of guidelines would be used to simplify the complexity of the optical matcher. First, all rules will be made equal in length. This implies breaking down the clauses into conjunctions of the same number of Boolean variables. Second, each clause in the rule is preferrably an atomic expression, i.e., a clause in a rule is not composed of other clauses. The implications of these two guidelines is that the rule memory is inefficiently implemented. The rows will not have the most compact representation but the associative match process will be very simple. Third, the conflict resolution that decides the rule to fire should be done in some simple fashion within the matcher. Otherwise, if the execution engine conducts the conflict resolution, all rules that could fire will have to be sent to it. Given the data handling capability of SPARO, this would not be advisable. Since the matcher will not possess much computing power, a simple scheme such as a 'first chosen' scheme may be preferred. In this scheme the first rule that is applicable will be chosen for execution.

8. SUMMARY AND CONCLUSIONS

We have developed an architecture (SPARO) for combinator graph reduction that is targeted to be implemented with emerging optical devices. We have shown that the constraints imposed by optical devices leads to a computer architecture with a very different configuration than those designed with electronic components.

The constraints imposed by optical devices and structures lead to a physically non-distributed architecture, which is partitioned functionally into a processor, a network processor, and a communications network. The nodes of this parallel architecture are fine-grained and employ optical memory in the form of registers to avoid the need for address decoding. The memory is not separated from the processing elements that implement combinator graph reduction. This has the advantage that the parallelism of a truly non-von Neumann processor is available.

In SPARO, the combinator graph is directly mapped onto the processor plane; each node of the processor contains one node of the combinator graph. The processor nodes perform the operations required for combinator graph reduction by sending messages that contain both information on the combinator graph to be modified and the corresponding instructions that need to be executed. This direct mapping, while exposing the maximum parallelism inherent in the problem, avoids the need for separate memory and processor. However, using a direct mapping means that SPARO may not efficiently use all its resources. Further, investigation of the processor operation is required to see if the use of fine-grained optical structures exposes enough parallelism to overcome the inefficiencies of the direct mapping of the combinator graph onto a processor.

We have shown that recursive and iterative control structures can be implemented as part of an optical computer. These basic control structures are present in almost all real symbolic processing application programs and must be available to the programmer. SPARO includes these as a basic part of its architecture without the use of stacks or specialized memory to manage the context switching. It achieves this by copying portions of the combinator graph. As with the direct mapping, this implementation needs futher investigation to determine if it is a viable approach for control structures.

Symbolic substitution and gate interconnects are the basic optical techniques required to implement SPARO. Symbolic substitution is used for control flow in each node and gated interconnects are used to move the data between registers and between nodes. These optical systems where chosen since they offer the greatest flexibility and are inherently digital. Examination of the number of symbolic substitution rules and the complexity of the interconnects will govern the cycle times that are possible for an optical computer such as SPARO. This in turn will determine if the use of fine-grained digital optical computers is a viable approach to optical computing.

REFERENCES

[1] Aloke Guha, Raja Ramnarayan, and Matthew Derstine, "Architectural Issues in Designing Symbolic Processors in Optics," to appear in the Proceedings, 14th International Symposium on Computer Architecture, June, 1987.

[2] Alexander A. Sawchuk and Timothy C. Strand, "Digital Optical Computing," Proceedings of the IEEE, Vol. 72, No. 7, July '84, pp. 758 - 779.

[3] Karl-Heinz Brenner, Alan Huang, Norbert Streibl, "Digital Optical Computing with Symbolic Substitution," Applied Optics, Vol. 25, September 1986, pp. 3054 - 3060.

[4] M. J. Murdocca, "Digital Optical Computing with One-Rule Cellular Automata," Tech. Report, AT&T Bell Labs, 1986.

[5] Rodney A. Schmidt and W. Thomas Cathey, "Optical Representations for Artificial Intelligence Problems," SPIE Vol. 625, Optical Computing (1986) pp. 226 - 233.

[6] Cardinal Ward and James Kottas, "Hybrid Optical Inference Machines: Architectural Considerations," Applied Optics, Vol. 25, March 1986, pp. 940-947.

[7] P. W. Smith and W. J. Tomlinson, "Bistable Optical Devices promise Subpicosecond Switching," IEEE Spectrum, Vol. 18, June 1981, pp. 26 - 33

[8] David Turner, "A New Implementation Technique for Applicative Languages," Software-Practice and Experience, Vol. 9, 1979, pp. 31 - 49.

[9] W. Daniel Hillis, The Connection Machine, MIT Press, 1985.

[10] Chris Clack and Simon L. Peyton Jones, "The Four-Stroke Reduction Engine," Proc. of the 1986 ACM Symposium on LISP and Functional Languages, pp. 220 - 232.

[11] David Turner, "Combinator Reduction Machines," Proc. of the International Workshop on High-Level Computer Architecture, University of Maryland, May 1984, pp. 5.26 - 5.38.

[12] Kenneth R. Traub, "An Abstract Parallel Graph Reduction Machine," Proc. of the 12th International Symposium on Computer Architecture, June 1985.

[13] Communications of the ACM, Special Issue on Architectures for Knowledge-Based Systems, Vol. 28, No. 9, September 1985.

[14] S. R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," IEEE Transactions on Computers, December 1984, pp. 1050 - 1071.

[15] M. Derstine, "Fundamental Physical Limits of Generalized Optical Interconnects," Honeywell Tech. Report, 1986.

APPENDIX: MACRO-INSTRUCTIONS

This appendix provides a description of the macro-instructions mentioned in Section 5. We first outline the additional flags and fields required for combinator and function reduction. The first few macro-instructions, sendup1 to S3, are common to the set of macro-instructions that are described thereafter. All macros are written in PASCAL-like statements, especially the conditional statements. This has been done purely for readability, and should not be interpreted to be indicative of the implementation. Note that statements between asterisks (*) represent comments.

Additional processor fields and flags (1-bit fields) required for reduction:

Type fields:

1) node_type = { regular, &, leaf }
2) combinator_level = { 1,2,3 }

Flags:

1) 2_parents_present
2) 2nd_apply_recvd
3) doing_S
4) awaiting_arity_incr_replies (* used when combinator_level = 2 *)
5) awaiting_eval_completes (* used for evaluating binary funcs *)

Data Fields:

1) temp (* holds the pointer to 'x' in the top node of the S mutation *) .fi
2) func (* used for indicating functions when evaluating unary or binary functions *)

Flags 3, 4, and 5 can be merged into a single flag, while
a single data field can be used for both data fields 1 and 2.

Types of atomic data:

CombOp_set:
combinators - S, B, C, K, I
unary & binary operators

Value_set:
any atom not in above set.

Three common message constructs are used by many macro-instructions. Note that the message fields are assigned only when the send instruction is invoked.

msg_normal:

```
out_buff/instr <- eval_complete
out_buff/data  <- RC
```

```
msg_mut_arg1:

    out_buff/instr <-  S/B/C/K/binary_func/1
    out_buff/data  <-  RC


msg_mut_arg2:

    out_buff/instr <-  S/B/C/2
    out_buff/data1 <-  LC
    out_buff/data2 <-  RC
```

The following is a list of macro-routines that would be necessary for graph reduction. Only combinator and function evaluations are considered. In case of function evaluation, the generic procedures to execute both unary and binary functions are provided.

Actions (macro-routines) taken upon completion of evaluation of a node:

```
sendup1: (* for a node having 1 or 2 parents: used to send messages to parent
         node and update state of the current node *)

         send msg to P1 (* msg is one of the three messages described above *)

         if not 2pp
         then
              load State <- dealloc
         else
              if 2nd_apply_recvd
              then
                   send msg to P2
                   load State <- dealloc
              else
                   load State <- evd


sendup2: (* for a node known to have 2 parents; used in increasing the arity of
         arguments f and g in S, B, and C combinators *)
     send msg to P1
     if 2nd_apply_recvd
     then
          send msg to P2
          load State <- dealloc
     else
          load State <- evd
```

```
in_buff/instr : arity_incr        (* used when the number of parents of a node has
                                     to be increased *)
```

```
State      : -

if 2_parents_present
then
   send
      out_buff/dest  <-  ?
      out_buff/instr <-  and_node_alloc
      out_buff/data  <-  in_buff/source
else
   load          2_parents_present <- true
   send
      out_buff/dest  <-  in_buff/source
      out_buff/instr <-  arity_incr_reply
      out_buff/data  <-  node_id
```

```
in_buff/instr : and_node_alloc        (* used for & node allocation when the number
                                         of parents of a node is increased (arity
                                         increase) *)

Status    : free

load_||    RC <- in_buff/source, node_type <- '&'
                                   (* load in parallel *)
   2_parents_present <- true,
send
   out_buff/dest  <-  in_buff/data1
   out_buff/instr <-  arity_incr_reply
   out_buff/data  <-  in_buff/source
```

```
in_buff/instr : arity_incr_reply      (* occurs at the node which sent out the  arity
                                         increase message *)

State      : evg

case
   (combinator_level = 1):
      load  RC <- in_buff/source
      sendup2:
   (combinator_level = 2):
      if (in_buff/data = LC)
         then load  LC <- in_buff/source
         else load  RC <- in_buff/source
      if awaiting_arity_incr_replies
      then
         load  awaiting_arity_incr_replies <- false
      else
```

```
        sendup2:
    (combinator_level = 3):
       load  temp <- in_buff/source
       send
          out_buff/dest  <-  ?
          out_buff/instr <-  S3
          out_buff/data1 <-  RC
          out_buff/data2 <-  temp
-----------------------------------------------------------------

in_buff/instr : apply        (* for regular and & nodes where message comes in after
                              evaluation is complete *)


 State       : evd

 send msg to P2
 load State <- dealloc
-----------------------------------------------------------------

in_buff/instr : apply        (* for regular and & nodes where the second apply arrives
                              while in the evaluating state *)


 State       : evg

  load P2 <- in_buff/source
  load 2_parents_present <- true
-----------------------------------------------------------------

in_buff/instr : apply_kill        (* for regular and & nodes where an apply_kill
                                  arrives while in the evaluating state *)


 State       : evg

 load 2_parents_present <- false      (* 2_parents_present must have been true *)
-----------------------------------------------------------------

in_buff/instr : apply_kill        (* for & nodes where an apply_kill arrives after the
                                  evaluation is complete *)


 State       : evd

 if (RC = ptr)
      then send apply_kill to RC
 if (LC = ptr)  (* i.e. msg to be sent is msg_mut_arg2 *)
    send apply_kill to LC
 load  State <- dealloc
-----------------------------------------------------------------
```

```
in_buff/instr : S3      (* this creates a new node setting LC, RC to incoming values,
                           and sends a reply back to the parent node *)

 Status     : free

 load           P1 <- in_buff/source
 load_||        LC <- in_buff/data1, RC <- in_buff/data2  send
    out_buff/dest  <-  P1
    out_buff/instr <-  S3_reply
-----------------------------------------------------------------
-----------------------------------------------------------------

in_buff/instr : apply  (* for all nodes receiving applies *)

 State      : nev

 load             P1 <- in_buff/dest
 case node_type of:
    regular node:
       send
          out_buff/dest  <-  LC
          out_buff/instr <-  apply
       load             State <- evg
    & node:
       send
          out_buff/dest  <-  RC
          out_buff/instr <-  apply
       load             State <- evg
    leaf node:
       load msg <- msg_normal
       sendup1
-----------------------------------------------------------------

in_buff/instr : S3_with_apply   (* creates a new node setting LC, RC to incoming
                                   values; assumes that an apply instruction was
                                   received *)

 Status     : free

 load           P1 <- in_buff/source
 load_||        LC <- in_buff/data1, RC <- in_buff/data2  send
    out_buff/dest  <-  LC
    out_buff/instr <-  apply
 load             State <- evg
-----------------------------------------------------------------

 in_buff/instr : apply_kill  (* to deallocate a leaf node *)

 State      : evd
```

```
  if 2_parents_present
  then
     load 2_parents_present <-false
  else
     load  State <-dealloc
-----------------------------------------------------------------

in_buff/instr : eval_complete   (* for a regular node receiving a completion
                                message *)


 State       : evg

 case in_buff/data in
    CombOp_set:
       load LC <- in_buff/data
       case LC of
         unary_func:
             goto    _evalng_unary_func
         S/B/C/K/binary_func:
             goto   combI
         I:
             goto   _I
    Value_set:
        if in_buff/source in functional_unit_set       (* function evaluated *)
         then
            load        RC <- in_buff/data
            load        node_type <- leaf
            load        msg <- msg_normal
            sendup1
        else
           if awaiting_eval_completes
             then
                load LC <- in_buff/data
                load  awaiting_eval_completes <- false
             else
                load RC <- in_buff/data
                send
                   out_buff/dest     <-      functional_unit
out_buff/instr <-  func
                   out_buff/data1 <-  LC
                   out_buff/data2 <-  RC
-----------------------------------------------------------------
: _evalng_unary_func       (* for evaluating an unary function subgraph *)

 load     func <- in_buff/data1
 load     LC <- nil

 send
    out_buff/dest  <-  RC
    out_buff/instr <-  apply
 load State <- evg
```

```
---------------------------------------------------------------
: comb1  (* actions taken at level 1 of a combinator subgraph *)

 load combinator_level <-1
 load msg <- msg_mut_arg1

   (* sending messages *)
 if not 2_parents_present
   then:
      do send msg to P1
   else:
       send
          out_buff/dest  <-  RC
          out_buff/instr <-  arity_incr

   (* setting states *)
 if not 2_parents_present
   then:
      load State <- dealloc
   else:
      load State <- evg
---------------------------------------------------------------
: _I  (* actions for I combinator reduction *)

 load     node_type <- '&'
 send
    out_buff/dest  <-  RC
    out_buff/instr <-  apply
 load State <- evg

---------------------------------------------------------------

in_buff/instr : K1     (* second instruction in sequence for K combinator in a regular
                       node at combinator_level 2 *)

 State    : evg

 send
    out_buff/dest  <-  RC
    out_buff/instr <-  apply_kill

 load      RC <- in_buff/data1
 goto      _I
---------------------------------------------------------------

in_buff/instr : S/B/C/1   (* first instruction in sequence for S, B, C combinator
                          reduction in regular node at combinator_level 2 *)

 State    : evg

 load     LC <- in_buff/data1
 goto     comb2
```

```
---------------------------------------------------------------
: comb2  (* actions taken at level 1 of a combinator subgraph *)

 load combinator_level <- 2
 load msg <- msg_mut_arg2

 load      awaiting_arity_incr_replies <- true

   (* sending messages *)
 if not 2_parents_present
   then:
       send msg to P1
   else:
       send
          out_buff/dest  <-  LC
          out_buff/instr <-  arity_incr
       send
          out_buff/dest  <-  RC
          out_buff/instr <-  arity_incr

   (* setting states *)
 if not 2pp
   then:
      load State <- dealloc
   else:
      load State <- evg
---------------------------------------------------------------

in_buff/instr : binary_func1     (* first instruction in sequence for evaluating a
                                 binary function in a regular node at top level of
                                 combinator subgraph *)


 State    : evg

 load     func <- in_buff/instr
 load     LC <- in_buff/data1

 load  awaiting_eval_completes <- true
 send
    out_buff/dest  <-  LC
    out_buff/instr <-  apply
 send
    out_buff/dest  <-  RC
    out_buff/instr <-  apply
 load State <- evg
---------------------------------------------------------------

in_buff/instr : S/B/C/2   (* second instruction in sequence for S, B, C combinator
                          reduction in regular node at combinator_level 3 *)

 State    : evg
```

```
load  combinator_level <- 3
load  ||        LC <- inbuff/data1, RC <- in_buff/data2, temp <- RC,
case S2:
      load      doing_S <- true
      send
         out_buff/dest  <-  temp
         out_buff/instr <-  arity_incr
case B2:
      send
         out_buff/dest  <-  ?
         out_buff/instr <-  S3    (* to allocate new node, and set LC and RC *)
         out_buff/data1 <-  RC
         out_buff/data2 <-  temp
case C2:
      send
         out_buff/dest  <-  ?
         out_buff/instr <-  S3_with_apply        (* same as S3 but also evaluate
                                                  LC *)
         out_buff/data1 <-  LC
         out_buff/data2 <-  temp
-----------------------------------------------------------------

in_buff/instr : S3_reply    (* response to S3 in a regular node at combinator_level 3
                            *)

 State    : evg

 load   RC <- in_buff/source
 if doing_S (* set earlier *)
    then
       send
          out_buff/dest  <-  ?
          out_buff/instr <-  S3_with_apply
          out_buff/data1 <-  LC
          out_buff/data2 <-  temp
    else (* doing_B *)
       send
          out_buff/dest  <-  LC
          out_buff/instr <-  apply
-----------------------------------------------------------------

in_buff/instr : any instruction other than apply        (* only for & nodes *)

 State : evg

 (* latch incoming message *)
 case in_buff/instr of:
    eval_complete: (* i.e. incoming data = atom *)
       load     RC <- in_buff/data1
       load     msg <- msg_normal
       load     node_type <- leaf
```

```
   sendup1:
S/B/C/K/binary_func/1 :
   load     RC <- in_buff/data1
   goto comb1
S/B/C/2  :
   load     LC <- in_buff/data1, RC <- in_buff/data2      goto comb2
```

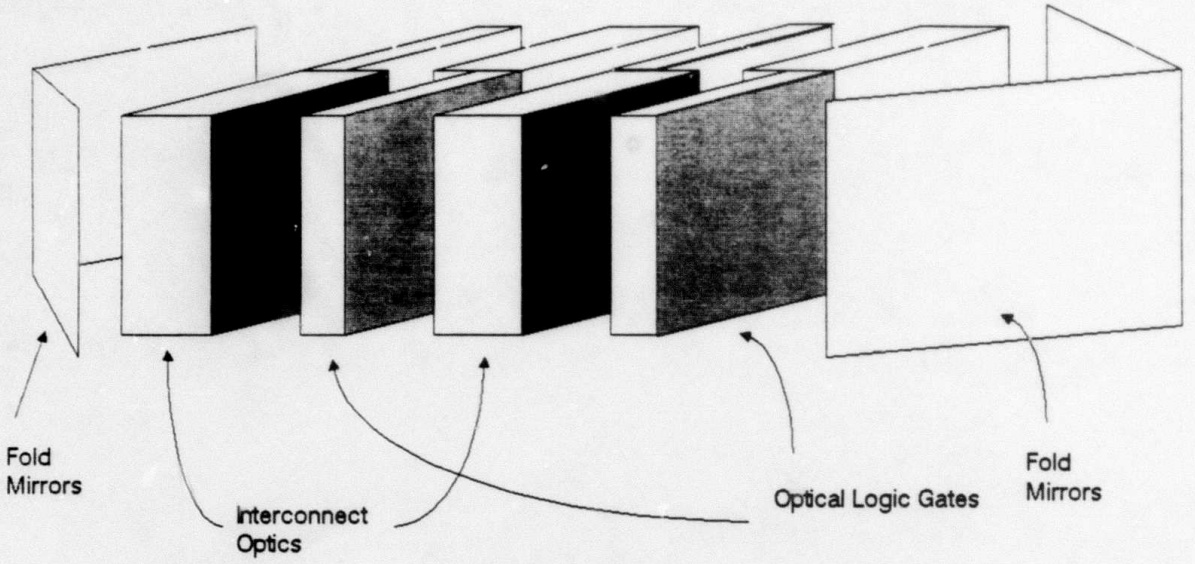


Figure 2.1 Schematic Layout of SPARO

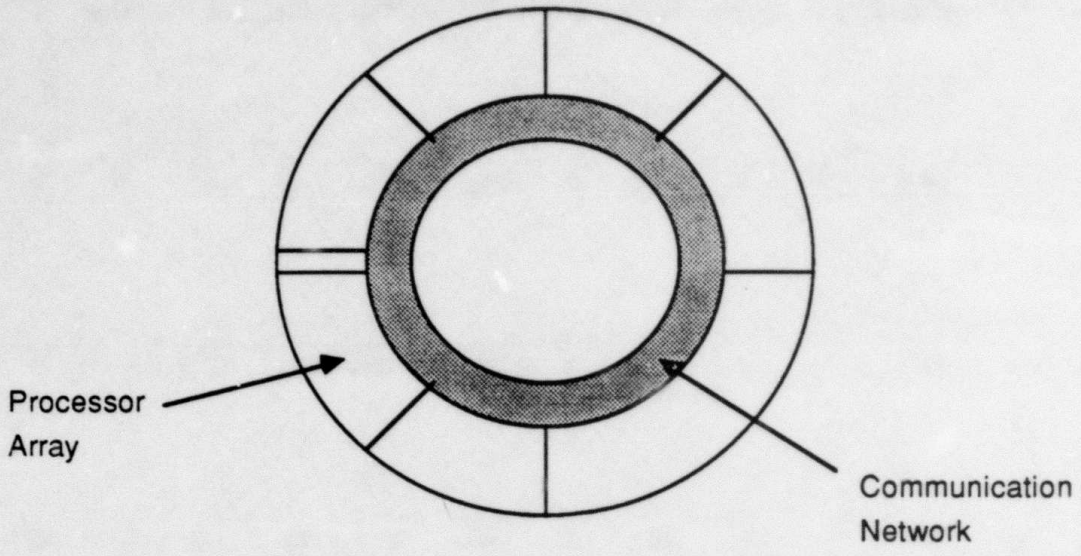
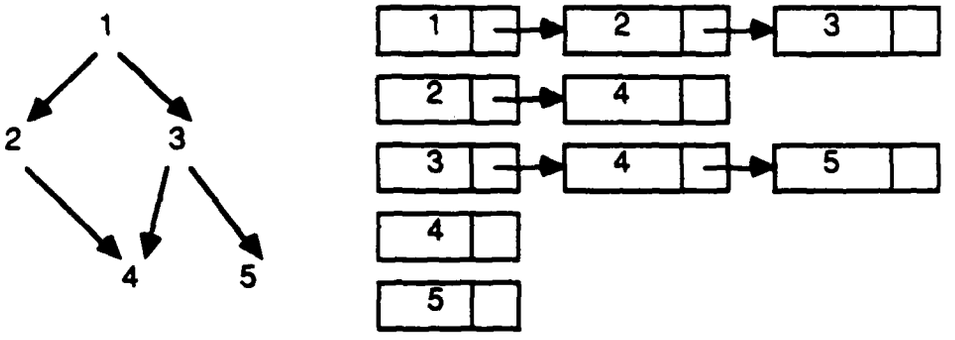


Figure 2.2 Scalable configuration for SPARO

a)



b)

1	2	3
2	-	4
3	4	5
4	-	-
5	-	-

Figure 3.1 a) Adjacency list representation of a graph
b) SKI combinator graph reduction

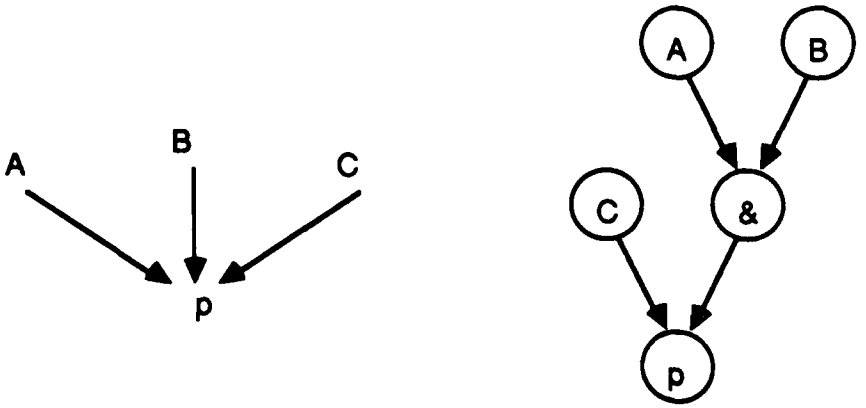


Figure 3.2 Using '&' nodes to generate binary graphs

Figure 4.2 Message format for SPARO

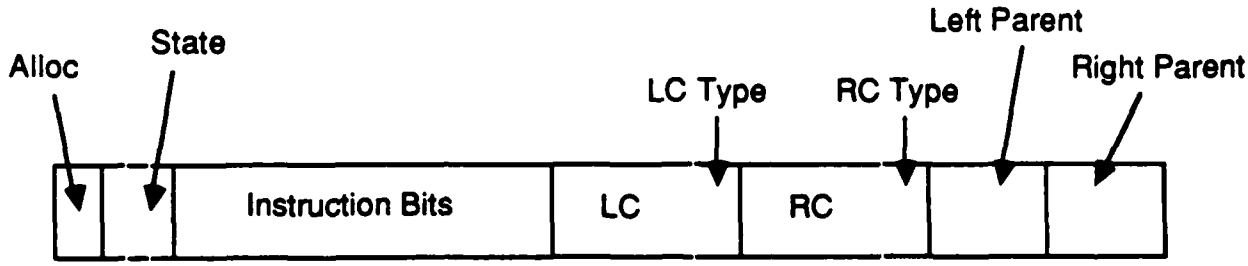


Figure 4.1 Row format for SPARO

S f g x -> f x (g x)

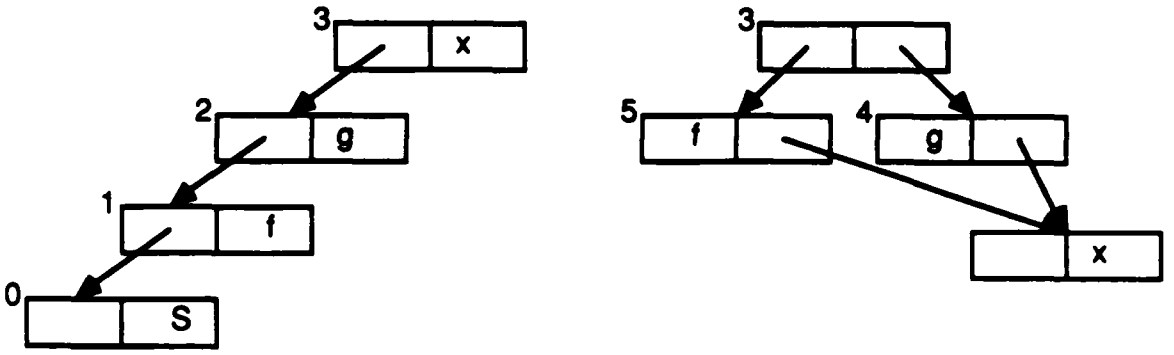


Figure 5.1 Combinator graph and reduction for 'S'

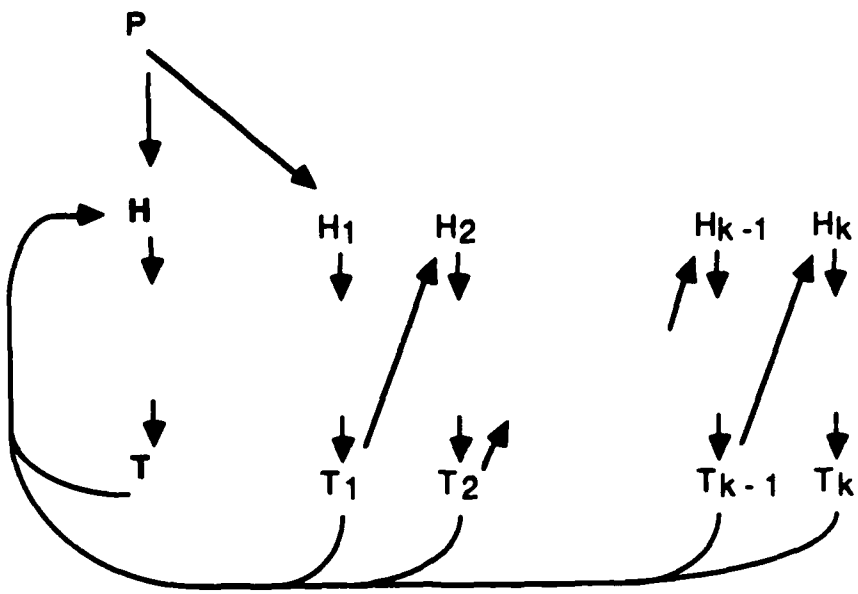


Figure 5.2 Expanding a recursive call

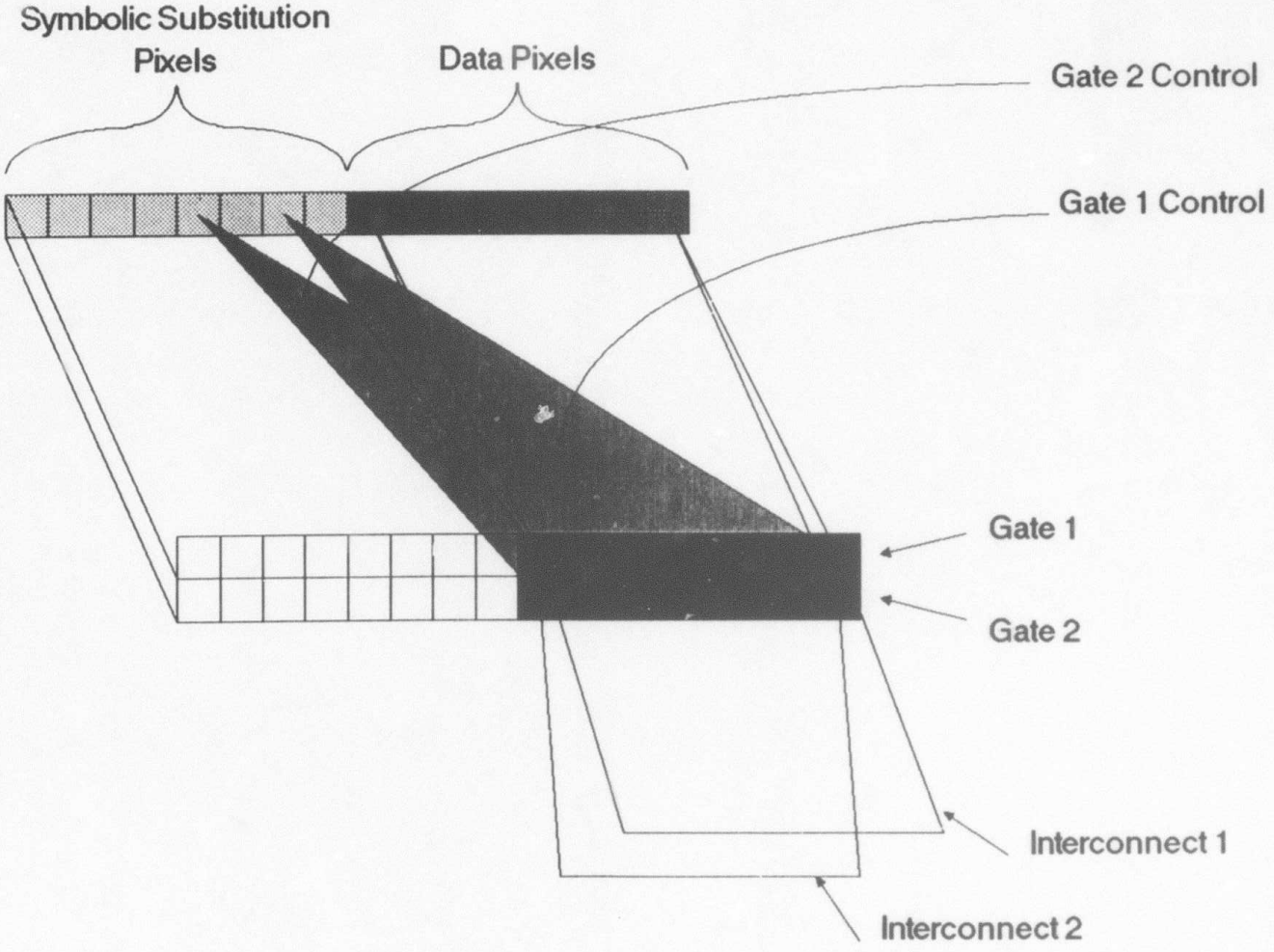


Figure 6.1 Combination of Symbolic Substitution and Gateable Interconnects

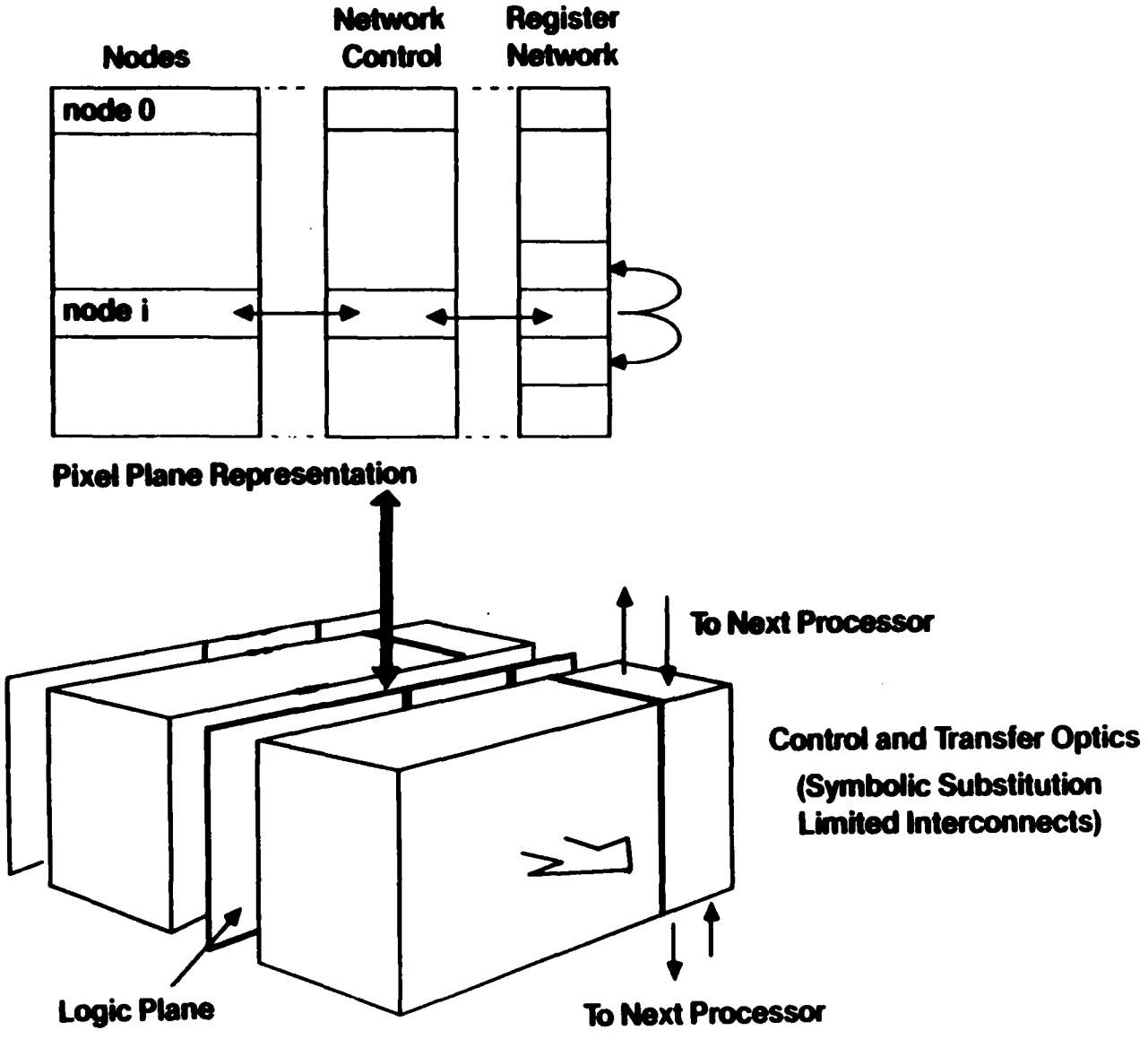


Figure 6.2 Internal Layout of SPARO

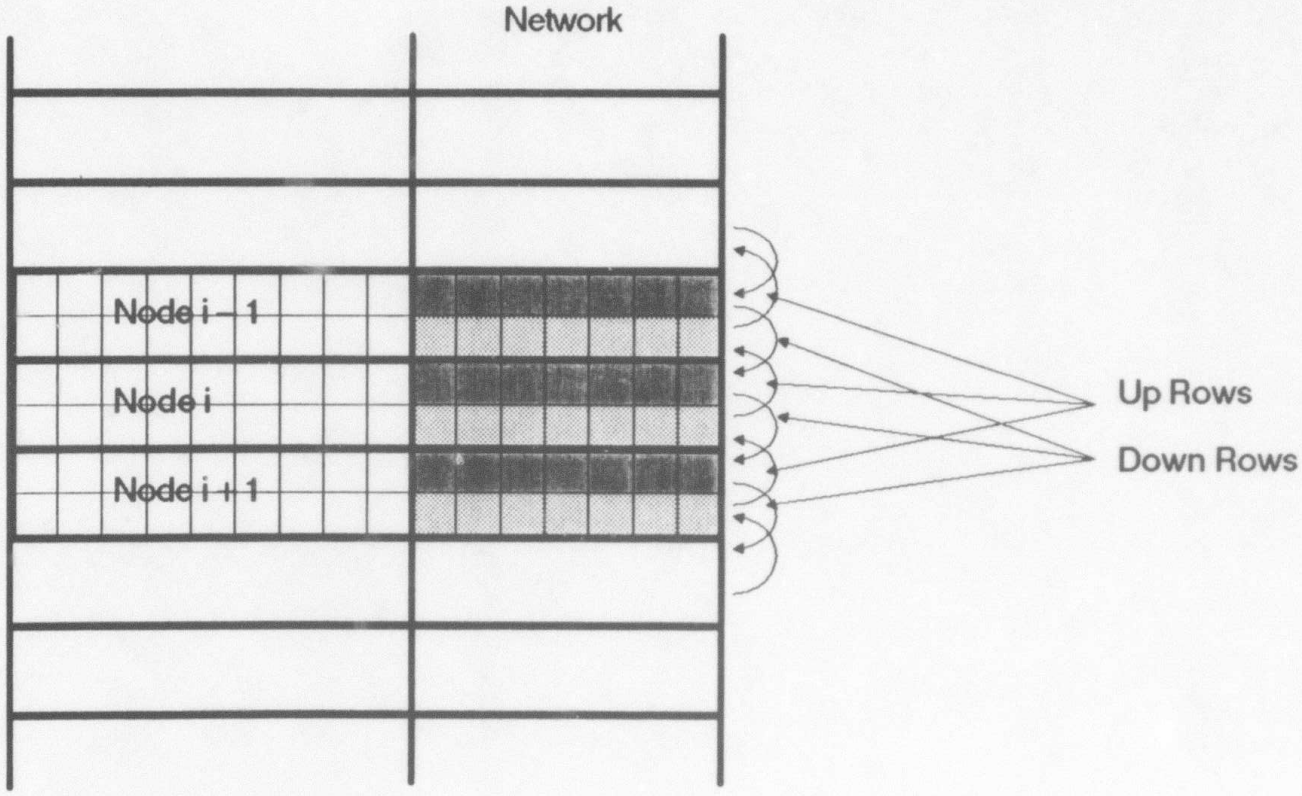


Figure 6.3 Network configuation of SPARO

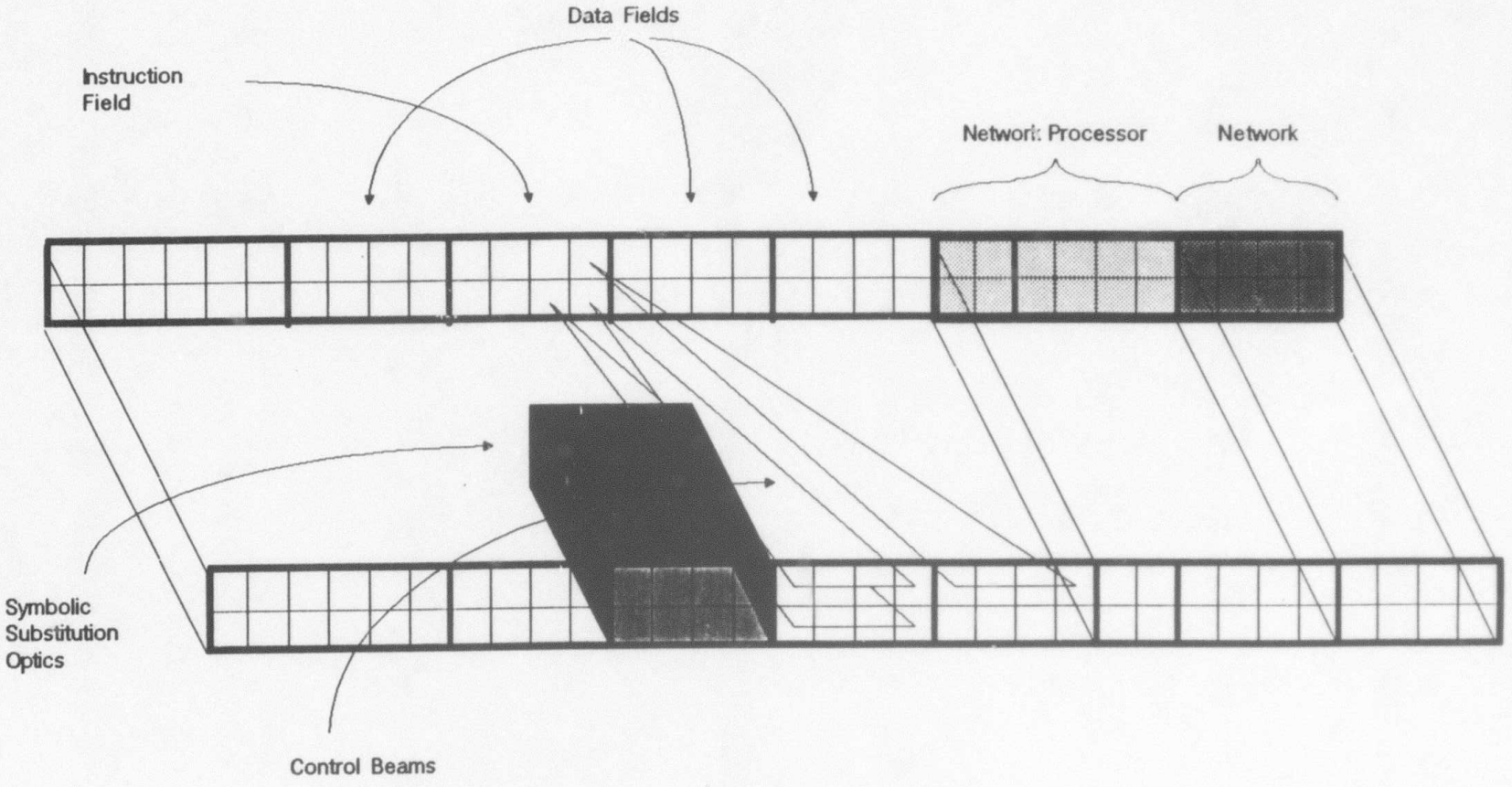


Figure 6.4 Basic Functionality of the optics of SPARO

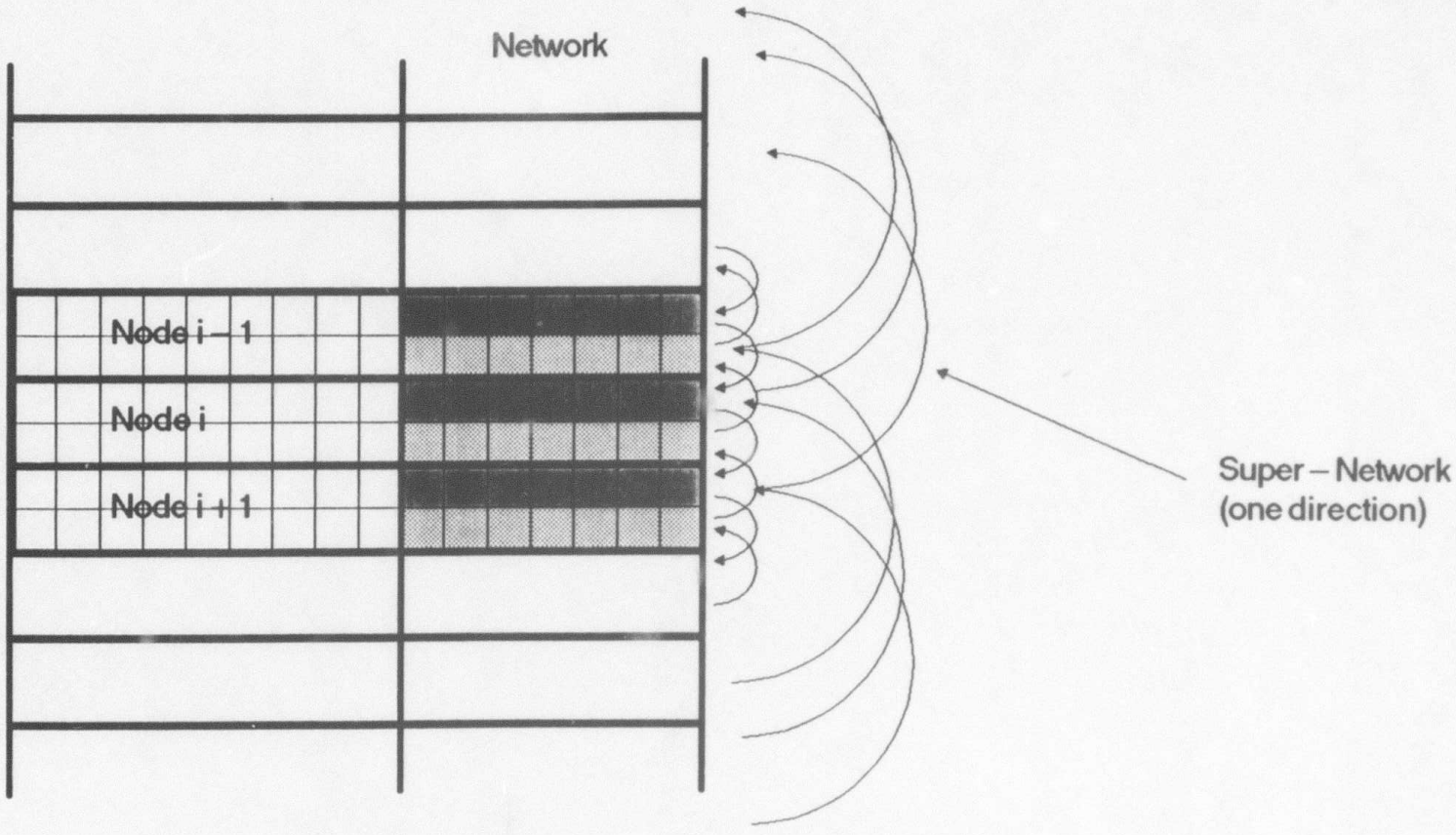


Figure 7.1 Addition of a Super Network to SPARO